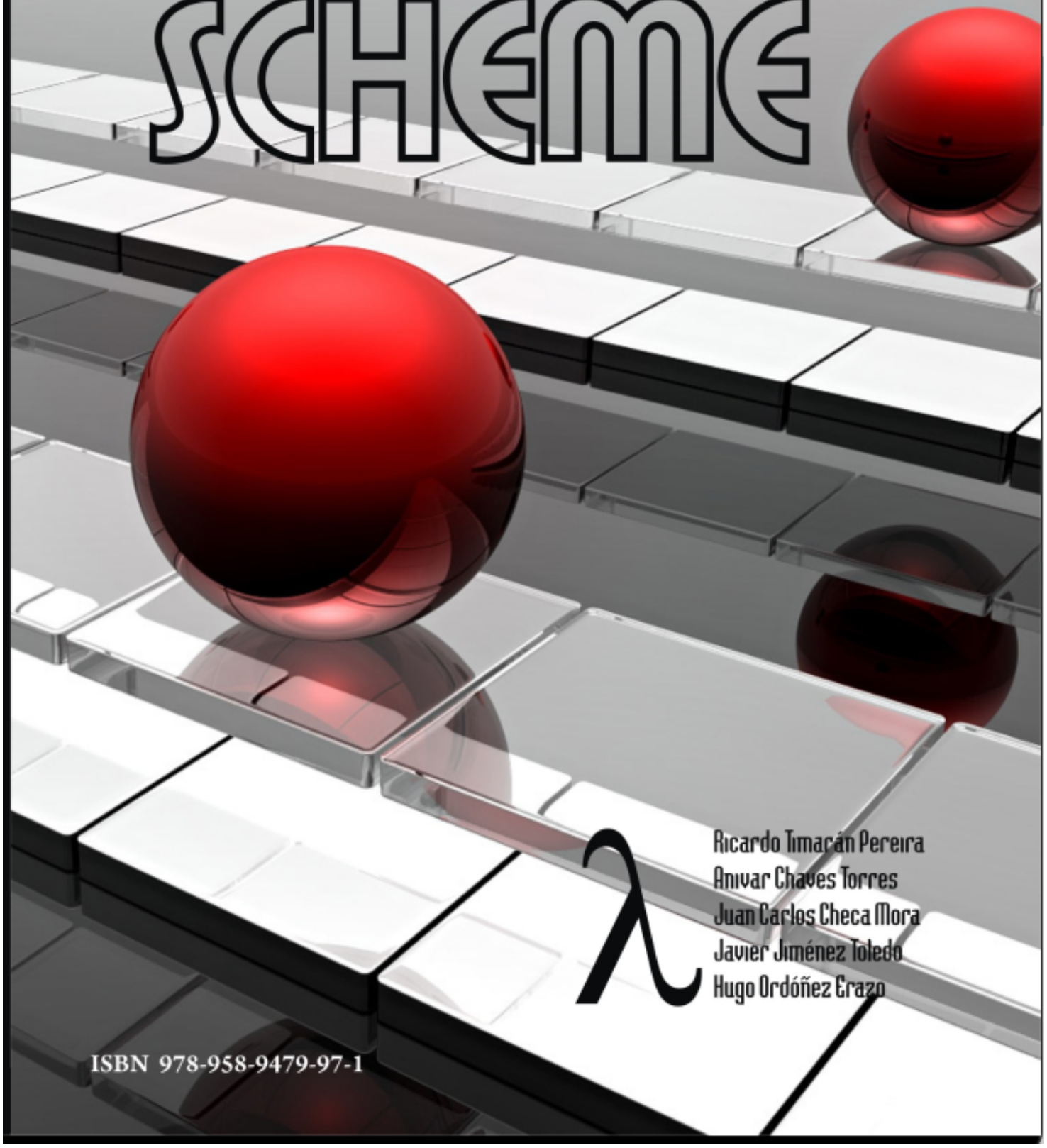
Introducción a la Programación con SCHEME



Ricardo Timarán Pereira
Anivar Chaves Torres
Juan Carlos Checa Mora
Javier Jiménez Toledo
Hugo Ordóñez Erazo

ISBN 978-958-9479-97-1

Introducción a la Programación con Scheme

Ricardo Timarán Pereira
Aníval Chaves Torres
Juan Carlos Checa Mora
Javier Jiménez Toledo
Hugo Ordóñez Erazo

Red Universitaria de Investigación en Sistemas de Nariño
RUISNAR

Red de Universidades Regionales Latinoamericanas
UREL
Capítulo Nariño

Copyright © 2009, RUISNAR

DERECHOS RESERVADOS

Se permite copiar y utilizar la información de este libro con fines académicos e investigativos siempre que se reconozca el crédito a los autores.

Se prohíbe la reproducción total o parcial por cualquier medio con fines comerciales sin autorización de los autores.

El pensamiento que se expresa en esta obra es responsabilidad exclusiva de los autores y no compromete la ideología de las instituciones que la financiaron.

ISBN: 978 – 958 – 9479 – 97 · 1

Edición y diagramación: Anivar Chaves Torres

Impreso por: Editorial Universidad de Nariño

A mi querida esposa Amanda
y a mis hijas Andrea, Sofía y Nataly.
Ricardo

A Diana Sofía, Jacky y Camilo
por su confianza y su comprensión.
Anívar

A mi hijo Julián Antonio
de quien aprendo, a quien enseño
y con quien construyo un nuevo mundo.
Juan Carlos

a Dios por ser mi guía,
a Davis, Santiago y Alejandro
quienes le dan sentido a mi vida,
a Ximena por su leal compañía,
a mis padres y hermanos
por su incondicional apoyo.
Javier

A DIOS,
fuente de toda sabiduría y conocimiento
y a mi madre por darme la vida.
Hugo

RED DE UNIVERSIDADES REGIONALES LATINOAMERICANAS
UREL
Capítulo Nariño

Carlos Folleco Erazo
Rector Universidad Cooperativa de Colombia
Presidente Red UREL Capítulo Nariño

Padre Evaristo Acosta Maestre
Rector Institución Universitaria CESMAG

Jaime Tito Colunge Benavides
Rector Corporación Universitaria Autónoma de Nariño

Luis Enrique Ortega Ruales
Director Zonal Universidad Nacional Abierta y a Distancia – UNAD

Orlando Rodríguez García
Director Fundación Universitaria San Martín – Sede Pasto

Deyanira Ortega
Directora Universidad Antonio Nariño – Sede Pasto

Gonzalo Hernández Arteaga
Director Regional Universidad Javeriana

Sara Ángela Arturo González
Directora Regional Servicio Nacional de Aprendizaje – SENA

Silvio Sánchez Fajardo
Rector Universidad de Nariño

Hermana Martha Estela Santa Castrillón
Rectora Universidad Mariana

Melba Rivas Benavides
Directora Universidad Santo Tomás Sede Pasto

Luis Antonio Heredia
Coordinador Fundación Universitaria del Área Andina Sede Pasto

Rocío de la Espriella Guerrero
Directora Escuela Superior de Administración Pública – ESAP

Roberto Narváez Moreno
Director Corporación Universitaria Remington – Sede Pasto

Gerardo Mesías Méndez
Director Ejecutivo Cámara de Comercio de Pasto

RED UNIVERSITARIA DE INVESTIGACIONES
RED UREL
Capítulo Nariño

Isabel Hernández Arteaga
Directora de Investigaciones
Universidad Cooperativa de Colombia
Coordinadora Red Universitaria de Investigaciones

Edmundo Apraez Guerrero
Vicerrector de Investigaciones, Postgrados
y Relaciones Internacionales
Universidad de Nariño

María Eugenia Córdoba
Vicerrectora de Investigaciones
Institución Universitaria CESMAG

Roberto García Castaño
Director del Centro de Investigaciones y Publicaciones
Universidad Mariana

Julio Rafael Burbano Erazo
Director de Investigaciones
Corporación Universitaria Autónoma de Nariño

Elehonora Argoty Pérez
Coordinadora de Investigaciones
Fundación Universitaria San Martín

Omar Martínez Roa
Coordinador Zonal de Investigaciones
Universidad Nacional Abierta y a Distancia – UNAD

RED UNIVERSITARIA DE INVESTIGACIÓN EN SISTEMAS DE NARIÑO
RUISNAR

Ricardo Timarán Pereira
Doctor en Ingeniería
Master of Science en Ingeniería
Especialista en Multimedia
Ingeniero de Sistemas y Computación
Universidad de Nariño
Director RUISNAR

Anívar Chaves Torres
Especialista en Docencia Universitaria
Ingeniero de Sistemas
Investigador
Institución Universitaria CESMAG

Juan Carlos Checa Mora
Especialista en Multimedia
Ingeniero de Sistemas
Investigador
Universidad Cooperativa de Colombia

Constanza Colunge
Ingeniera de Sistemas
Investigadora
Corporación Universitaria Autónoma de Nariño

Javier Jiménez Toledo
Especialista en Docencia Universitaria
Ingeniero de Sistemas
Investigador
Universidad de Nariño

Hugo Ordóñez Erazo
Especialista en Gerencia Informática
Ingeniero de Sistemas
Investigador
Universidad Mariana

CONTENIDO

PRÓLOGO **21**

CAPÍTULO 1
PARADIGMAS DE PROGRAMACIÓN **25**
1.1 EL CONCEPTO DE PARADIGMA DE PROGRAMACIÓN 25
1.2 TIPOS DE PARADIGMAS DE PROGRAMACIÓN 27
1.2.1 Paradigma Imperativo 27
1.2.2 Paradigma Declarativo 31
1.3 PARADIGMA FUNCIONAL 36
1.3.1 Historia 36
1.3.2 Programación funcional 37
1.3.3 Lenguajes Funcionales 37

CAPÍTULO 2
FUNDAMENTOS DEL LENGUAJE SCHEME **41**
2.1 HISTORIA 41
2.2 CARACTERÍSTICAS DEL LENGUAJE SCHEME 43
2.3 ESTRUCTURA DE UN PROGRAMA EN SCHEME 44
2.4 ELEMENTOS DEL LENGUAJE SCHEME 46
2.4.1 Comentarios 46
2.4.2 Tipos de datos 46
2.4.3 Variables 52
2.4.5 Expresiones 55
2.4.6 Funciones 57

CAPITULO 3
ENTRADA Y SALIDA DE DATOS **61**
3.1 ESCRITURA DE DATOS 62
3.1.1 Función *display* 62
3.1.2 Función *write* 63
3.1.3 Función *newline* 65
3.2 LECTURA DE DATOS 65
3.2.1 Función *read* 66

3.2.2 Función *read-char* 67
3.2.3 Función *read-line* 69

CAPÍTULO 4
DECISIONES 75
4.1 LAS DECISIONES EN PROGRAMACIÓN 75
4.2 TIPOS DE DECISIONES 76
4.2.1 Decisión simple 76
4.2.2 Decisión doble 77
4.2.3 Decisión múltiple 78
4.3 FUNCIONES PARA IMPLEMENTAR DECISIONES 80
4.3.1 Función *when* 80
4.3.2 Función *unless* 81
4.3.3 Función *if* 84
4.3.4 Función *case* 88
4.4 DECISIONES ANIDADAS 92
4.4.1 Anidamiento con *when*, *unless*, *if* y *case* 93
4.4.2 Función *cond* 94

CAPÍTULO 5
ITERACIONES 97
5.1 CONTADORES Y ACUMULADORES 98
5.1.1 Contadores 98
5.1.2 Acumuladores 98
5.2 FORMA ESPECIAL *LET* 99
5.3 EXPRESIÓN *DO* 105

CAPÍTULO 6
RECURSIVIDAD 111
6.1 LA RECURSIVIDAD COMO TÉCNICA DE PROGRAMACIÓN 112
6.2 ESTRUCTURA DE UNA FUNCIÓN RECURSIVA 113
6.3 EJECUCIÓN DE FUNCIONES RECURSIVAS 115
6.4 ENVOLTURAS PARA FUNCIONES RECURSIVAS 119
6.5 TIPOS DE RECURSIVIDAD 120
6.6 RECURSIVIDAD EN SCHEME 121
6.7 EJEMPLOS DE SOLUCIONES RECURSIVAS 123
6.7.1 Función sumatoria 123
6.7.2 Función factorial 125
6.7.3 Función potencia 125
6.7.4 Función fibonacci 128

6.7.5 Máximo Común Divisor 131
6.8 EFICIENCIA DE LA RECURSIVIDAD 131

CAPÍTULO 7
ARREGLOS 135
7.1 EL CONCEPTO DE VECTOR 136
7.2 DEFINICIÓN DE UN VECTOR EN SCHEME 136
7.3 FUNCIONES PARA MANEJO DE VECTORES 140
7.3.1 *make-vector* 140
7.3.2 *vector* 142
7.3.3 *list-vector* 144
7.3.4 *vector-length* 145
7.3.5 *vector-ref* 146
7.3.6 *vector-set!* 151
7.3.7 *vector-fill!* 162
7.3.8 *vector?* 163
7.4 BÚSQUEDA DE UN ELEMENTO EN UN VECTOR 164
7.5. IMPLEMENTACIÓN DE UN PROGRAMA DE INVENTARIO 166

CAPÍTULO 8
LISTAS 173
8.1 MANEJO DE LISTAS CON SCHEME 174
8.2 FUNCIONES PARA MANEJO DE LISTAS 177
8.2.1 Funciones *car* y *first* 179
8.2.2 Funciones *cdr* y *rest* 180
8.3.3 Función *length* 185
8.3.4 Función *cons* 187
8.3.5 Función *list* 192
8.3.6 Función *findf* 194

REFERENCIAS 197

LISTA DE CUADROS

Cuadro 2.1. Estructura básica de un programa en Scheme 45
Cuadro 2.2. Ejemplo de un programa en Scheme.....45
Cuadro 2.3. Uso de comentarios 46
Cuadro 2.4. Funciones tipos de datos.....51
Cuadro 2.5. Programa para evaluar tipos de datos.....51
Cuadro 2.6. Ejemplo de uso de variables.....55
Cuadro 2.7. Ejemplos de Notaciones 56
Cuadro 2.8. Ejemplos de expresiones 57
Cuadro 2.9. Ejemplos de funciones internas.....58
Cuadro 2.10. Definición de una función.....59
Cuadro 2.11. Función para sumar dos números.....59
Cuadro 3.1. Ejemplo de la función display. 63
Cuadro 3.2. Ejemplo de la función *write*.....64
Cuadro 3.3. Ejemplo de la función *read* 67
Cuadro 3.4. Ejemplo de la función *read-char* 68
Cuadro 3.5. Ejemplo de la función *read-line* 69
Cuadro 3.6. Programa para sumar dos números 70
Cuadro 3.7. Programa para calcular área y perímetro de un círculo ...72
Cuadro 3.8. Programa para calcular el valor futuro de una inversión..73
Cuadro 4.1. Función valor-absoluto.....77
Cuadro 4.2. Función par-impar.....78
Cuadro 4.3. Función arábigo-romano.....79

Cuadro 4.4 Función calcular-total 80
Cuadro 4.5. Función para calcular bonificación 81
Cuadro 4.6. Programa para calcular cuota moderadora 82
Cuadro 4.7. Función calcular-total-2 83
Cuadro 4.8. Función múltiplo 83
Cuadro 4.9. Función evalúa-divisor 84
Cuadro 4.10. Función auxilio-transporte 86
Cuadro 4.11. Función número-mayor 86
Cuadro 4.12. Función división 87
Cuadro 4.13. Programa para calcular áreas de triángulo y círculo 87
Cuadro 4.14. Función nombre-día 89
Cuadro 4.15. Función número-días 90
Cuadro 4.16. Menú para las operaciones aritméticas básicas 91
Cuadro 4.17 Función año-bisiesto 93
Cuadro 4.18 Función número-mayor 94
Cuadro 4.19. Función nuevo-sueldo utilizando cond 95
Cuadro 4.20. Función nuevo-sueldo utilizando *if* 96
Cuadro 5.1. Ciclo con la forma especial *let* 99
Cuadro 5.2. Ejemplo de ciclo con *let* 100
Cuadro 5.3. Programa para entrar números y sumarlos 100
Cuadro 5.4. Programa para encontrar los divisores de un número 101
Cuadro 5.5. Programa para encontrar el promedio y los números mayor y menor de entre *n* entradas. 102
Cuadro 5.6. Programa para encontrar el promedio y los números mayor y menor de entre *n* entradas. 103
Cuadro 5.7. Programa para validar notas y calcular el promedio. 104
Cuadro 5.8. Programa para mostrar los números pares entre 2 y 20 106

Cuadro 5.9. Programa para mostrar la tabla de multiplicar de un número *n* 107
Cuadro 5.10. Programa para calcular promedio de notas 108
Cuadro 5.11. Función para encontrar el MCM de dos números 108
Cuadro 5.12. Programa para determinar si un número es primo. 109
Cuadro 6.1 Función factorial en pseudocódigo 114
Cuadro 6.2 Pseudo código del programa principal 119
Cuadro 6.3 Estructura de una función recursiva en Scheme 121
Cuadro 6.4 Función para calcular el factorial de un número 122
Cuadro 6.5 Envoltura de la función factorial 122
Cuadro 6.6 Función sumatoria 124
Cuadro 6.7 Programa recursivo para calcular sumatoria 124
Cuadro 6.8 Función potencia 127
Cuadro 6.9 Programa para calcular una potencia 127
Cuadro 6.10 Pseudocódigo de la función Fibonacci 129
Cuadro 6.11 Programa para calcular la serie Fibonacci 130
Cuadro 6.12. Función para encontrar el Máximo Común Divisor 132
Cuadro 7.1 Definición de vectores 138
Cuadro 8.2 Funciones para manejo de vectores 140
Cuadro 7.3 Creación de vectores con la función *make·vector* 142
Cuadro 7.4. Creación de vectores con la función *vector* 143
Cuadro 7.5 Programa para mostrar un vector posición por posición 148
Cuadro 7.6. Programa para mostrar un vector utilizando un ciclo 149
Cuadro 7.7. Programa para mostrar un vector recursivamente 149
Cuadro 7.8. Programa para mostrar un vector en orden inverso 151
Cuadro 7.9. Programa que captura datos y los guarda en un vector 153
Cuadro 7.10 Conteo de dígitos almacenados en un vector 155
Cuadro 7.11 Vector de vectores 159

Cuadro 7.12 Matriz de datos 161
Cuadro 7.13 Programa para determinar si un objeto es de tipo vector 163
Cuadro 7.14 Búsqueda lineal 165
Cuadro 7.15. Descuentos por cantidad 166
Cuadro 7.16. Programa para manejo de un inventario 168
Cuadro 8.1 Definición de listas 176
Cuadro 8.2 Funciones para manejo de listas 178
Cuadro 8.3. Programa que crea una lista de colores primarios 180
Cuadro 8.4. Programa ejemplo de *cdr* y *rest* 181
Cuadro 8.5 Impresión recursiva de una lista 182
Cuadro 8.6 Impresión inversa de una lista 183
Cuadro 8.7 Sumatoria de los elementos de una lista 184
Cuadro 8.8. Programa para encontrar el menor valor en una lista 185
Cuadro 8.9 Conteo de los elementos de una lista 187
Cuadro 8.10. Ejemplo de la función *cons* 188
Cuadro 8.11 Aplicación de la función *cons* 189
Cuadro 8.12 Creación de la lista estudiantes con la función *cons* 190
Cuadro 8.13. Adición de un número en una lista ordenada 191
Cuadro 8.14 Creación de listas con la función *list* 193
Cuadro 8.15. Ejemplo de función *findf* 194

LISTA DE FIGURAS

Figura 1.1. Historia de algunos de los lenguajes de programación 26
Figura 1.2. Clasificación básica de los paradigmas de programación .28
Figura 1.3 Categoría de los lenguajes funcionales.....39
Figura 2.1. Los tipos de datos en Scheme 47
Figura 2.2. Ejemplo de vector.....50
Figura 3.1. Cuadro de entrada de datos.....67
Figura 6.1 Esquema de una función recursiva..... 113
Figura 6.2. Rastreo de activación para la función factorial..... 117
Figura 6.3 Marcos de activación de la serie Fibonacci 130
Figura 8.1 Lista múltiplos de 3 174
Figura 8.2 Lista días-semana 175
Figura 8.3 Lista colores-primarios 179
Figura 8.4 Funciones *cdr* y *rest* 181
Figura 8.5 Lista de alimentos 186
Figura 8.6 Ejemplo de la función *cons* 188
Figura 8.7. Ejemplo de la función *list* 192

PRÓLOGO

"No basta saber,
se debe también aplicar.
No es suficiente querer,
se debe también hacer."
J. W. Goethe

Un curso de introducción a la programación tiene como objetivo brindar a los estudiantes los conceptos, métodos y técnicas, básicos y necesarios, para la solución de problemas utilizando un lenguaje de programación enmarcado dentro de un paradigma específico.

A los estudiantes les resulta muy difícil el paso del enunciado del problema al programa. La programación exige, inicialmente, el entendimiento del problema a través de su análisis; luego, la construcción y prueba del algoritmo y, finalmente, la codificación en un lenguaje de programación. En un primer curso de programación, la cuestión ha resolver es cuál es el paradigma y el lenguaje más adecuados para la enseñanza de los fundamentos de programación.

En la actualidad, existe un debate sobre la conveniencia de aplicar un enfoque estructurado, orientado a objetos o funcional para enseñar los fundamentos de la programación en Ingeniería de Sistemas. Los modelos estructurado y orientado a objetos, por ser imperativos, describen la programación en términos del estado del programa y sentencias que cambian dicho estado. El modelo funcional tiene como objetivo obtener programas donde se exprese la solución a través de funciones, en los que no es necesario bajar al nivel de la máquina para describir el proceso llevado a cabo por el programa, evitando el concepto de estado del cómputo.

La Red Universitaria de Investigación en Ingeniería de Sistemas de Nariño - RUISNAR, conformada por la Universidad de Nariño, Universidad Mariana, Universidad Cooperativa, Institución Universitaria CESMAG y la Corporación Universitaria Autónoma de Nariño, instituciones con sede en la ciudad de Pasto, capital del departamento de Nariño (Colombia) y con programas de Ingeniería de Sistemas, realizó el proyecto de investigación *Validación del modelo funcional en la enseñanza de la programación con el lenguaje Scheme en programas Ingeniería de Sistemas*, en el que se estudió, documentó, aplicó y evaluó el modelo funcional utilizando el lenguaje Scheme, en una de las asignaturas del componente de programación de Ingeniería de Sistemas, con fin de desarrollar en el estudiante la capacidad de entender y utilizar la recursividad como una herramienta para mejorar su desempeño en la construcción de programas que den solución efectiva a problemas específicos en el campo del desarrollo de software. Este proyecto fue financiado por la Red de Universidades Regionales Latinoamericanas -UREL - capítulo Nariño. Este libro es uno de los resultados de esta investigación.

Este libro se elaboró como texto guía para los estudiantes de las diferentes instituciones que participaron en este proyecto y está pensado como un texto guía de un curso introductorio de programación, para estudiantes sin conocimientos previos sobre programación. El contenido de este libro está diseñado para ser cubierto en un semestre académico y puede ser seguido de una asignatura de programación imperativa.

La organización de los capítulos llevará al lector a conocer desde los conceptos a las diferentes estructuras de programación, y todos los aspectos del lenguaje Scheme se estudian progresivamente con aplicaciones prácticas en Dr. Scheme.

El *capítulo 1* es una introducción a los diferentes paradigmas de programación, haciendo énfasis en el paradigma funcional.

El *capítulo 2* describe los fundamentos del lenguaje SCHEME, su historia, sus características, elementos y la estructura de un programa en SCHEME.

El *capítulo 3* trata las sentencias básicas de entrada y salida con SCHEME.

El *capítulo 4* aborda los diferentes tipos de estructuras condicionales que maneja el lenguaje SCHEME.

El *capítulo 5* estudia las estructuras repetitivas, comúnmente conocidas como ciclos y su forma de implementarlos con el lenguaje SCHEME.

El *capítulo 6* explica la recursividad como técnica de programación y la estructura y ejecución de funciones recursivas en SCHEME.

El *capítulo 7* aborda el manejo de arreglos, su definición y las funciones de SCHEME para su manipulación.

El *capítulo 8* termina el libro con el estudio de listas, su definición y manejo con el lenguaje SCHEME.

En todos los capítulos se incluyen varios ejemplos que permiten al lector un mayor entendimiento de los temas. Para facilitar la ubicación del código fuente, en el apéndice A se presenta una lista de ejemplos.

Los programas de ejemplo, el código fuente y documentación adicional sobre este tema puede consultarse en el sitio web:

http://virtual.iucesmag.edu.co/ruisnar/scheme

Agradecimientos

A la Red de Universidades Regionales Latinoamericanas UREL, capítulo Nariño por hacer posible la realización del proyecto de investigación y la publicación de este libro.

A Gonzalo Hernández, quien participó en las dos primeras fases de la investigación y colaboró en el estudio del lenguaje Scheme y en la capacitación de los investigadores de RUISNAR.

A Wilson Pantoja, quien participó en los primeros meses de investigación en representación de la Fundación Universitaria San Martín y colaboró en el estudio del editor PLT Dr. Scheme.

A Johana Muñoz, quien participó en la primera fase de investigación y colaboró en la revisión bibliográfica sobre los fundamentos del lenguaje Scheme.

A Constanza Colunge, quien participó en la tercera fase de la investigación en representación de la Corporación Universitaria Autónoma de Nariño.

A Mirian Benavides, Sonia Narváez y José María Muñoz por el tiempo que dedicaron a revisar el documento y por sus acertadas observaciones.

CAPÍTULO 1

PARADIGMAS DE PROGRAMACIÓN

Tan solo por la educación
puede el hombre llegar a ser hombre.
El hombre no es más que
lo que la educación hace de él.
Kant

Un paradigma está constituido por los supuestos teóricos generales, las leyes y las técnicas para su aplicación que adoptan los miembros de una determinada comunidad científica. Es un marco de referencia que impone reglas sobre cómo se deben hacer las cosas, indica qué es válido dentro de éste y qué está fuera de sus límites. Un paradigma distinto implica nuevas reglas, elementos, límites y maneras de pensar, o sea implica un cambio. Los paradigmas pueden ser considerados como patrones de pensamiento para la resolución de problemas.

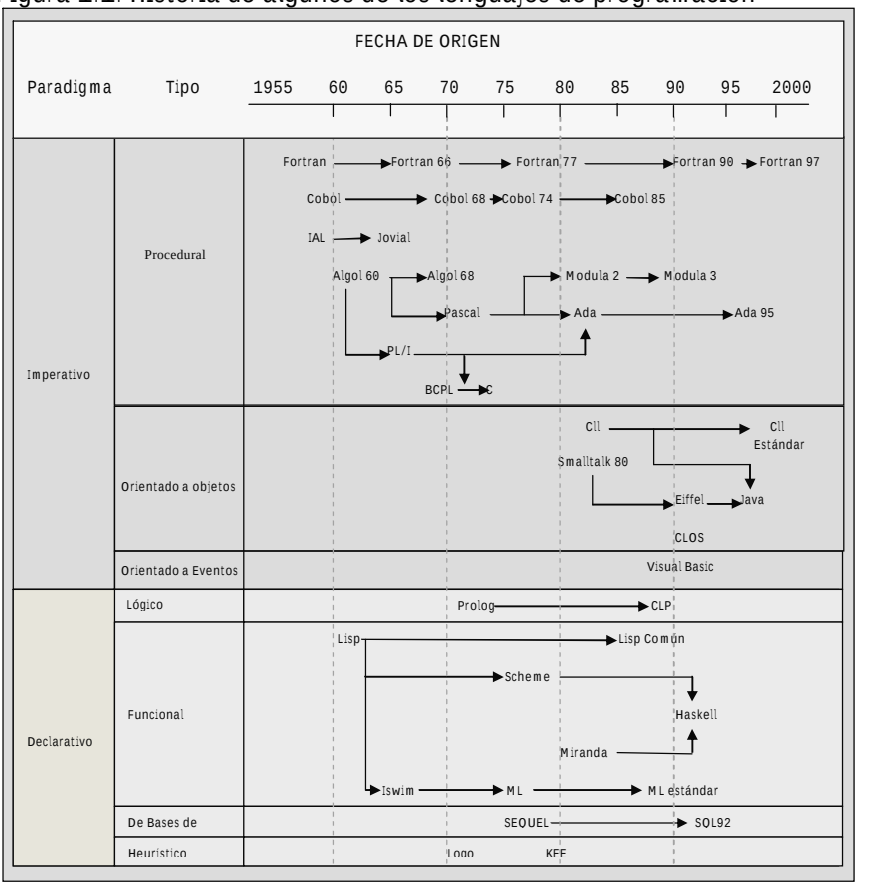
1.1 EL CONCEPTO DE PARADIGMA DE PROGRAMACIÓN

Un paradigma de programación es una colección de modelos conceptuales que orientan el proceso de diseño y desarrollo de programas con unas directrices específicas, tales como: estructura modular, fuerte cohesión y alta rentabilidad. La estructura conceptual de dichos modelos está pensada de modo tal que determinan la forma correcta de los programas y esquematizan la manera de pensar y formular soluciones que deben expresarse mediante un lenguaje de programación.

Los paradigmas de programación representan un enfoque particular o filosofía para la construcción del software. No es mejor uno que otro; sino que cada uno tiene ventajas y desventajas. También hay situaciones donde un paradigma resulta más apropiado que otro.

Un paradigma de programación esta asociado con cierta familia de lenguajes de programación (ver figura 1.1).

Figura 1.1. Historia de algunos de los lenguajes de programación



Muchos lenguajes se diseñan para poner un paradigma en ejecución, por ejemplo, Smalltalk y Java se asocian a la Programación Orientada a Objetos, mientras que Scheme y Haskell se asocian a la Programación Funcional. Otros lenguajes, tales como Common Lisp, Python, y Oz son para permitir el uso de paradigmas múltiples. (GNU Free Documentation License).

El desarrollo de los lenguajes de programación, sigue de cerca el desarrollo de los procesos físicos y electrónicos usados en las computadoras, muchos han tenido una vida y utilidad limitada, mientras que otros han tenido un éxito en su aplicación en uno o más dominios hasta el punto que han influido en el diseño de los lenguajes futuros. En la figura 1.1 se describe el resumen general de la historia de algunos de los lenguajes de programación que han tenido más influencia (Tucker y Noonann, 2003).

1.2 TIPOS DE PARADIGMAS DE PROGRAMACIÓN

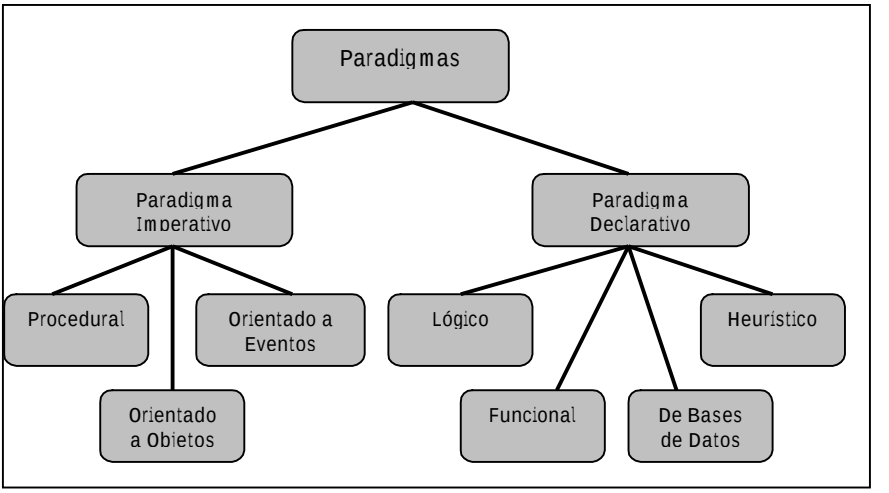
Dado que un paradigma es un conjunto de principios generales que un programador ha de seguir para desarrollar un programa, cada lenguaje de programación permite al programador seguir un paradigma determinado, por tanto los lenguajes de programación se pueden también clasificar según el paradigma que siguen.

Una clasificación básica de los paradigmas de programación se muestra en la figura 1.2.

1.2.1 Paradigma Imperativo

La *programación imperativa* recibe este nombre debido a que está basada en comandos que actúan sobre variables. Este paradigma es el más antiguo (1950s), los primeros diseñadores se dieron cuenta que la utilización del concepto de variable unido al comando de asignación constituían una abstracción simple pero muy efectiva para poder controlar la información almacenada en la memoria de un computador. (Serón, Magallón y Baldassarri, 2006)

Figura 1.2. Clasificación básica de los paradigmas de programación



Se determina también como un modelo abstracto que consiste en un gran almacenamiento de memoria donde la computadora guarda una representación codificada de un cálculo y ejecuta una secuencia de comandos que modifican el contenido de ese almacenamiento.

Un programa en un lenguaje imperativo aparece como una lista de instrucciones u órdenes elementales que han de ejecutarse una tras otra, en el orden en que aparecen en el programa. Las instrucciones de un programa imperativo utilizan datos almacenados en la memoria del computador, llamados variables. Para realizar algún cálculo, se parte de ciertos datos almacenados y se realizan diversas operaciones (instrucciones); al final, el resultado está almacenado en alguna celda de memoria.

A este tipo de paradigma de programación se le suele llamar algorítmico, dado que el significado de algoritmo es análogo al de receta, método, técnica, procedimiento o rutina, y se define como un conjunto finito de reglas diseñadas para crear una secuencia de operaciones para resolver un tipo específico de problemas. De esta forma para Wirth, un programa viene definido por la ecuación

Programas = Algoritmos + Estructura de Datos

Los elementos principales de la programación imperativa son:

- Variable. El componente principal es la memoria, compuesto por un gran número de celdas donde se almacenan los datos y que son referenciadas por medio de su nombre (variable). El conjunto de valores de todas las variables del programa en un momento dado representa el estado del programa.

- Operaciones de asignación. Cada valor calculado debe ser asignado a la variable mediante operaciones de asignación. De esta forma se modifica el estado del programa.

- Bifurcación. Es una estructura base de decisión en un lenguaje imperativo.

- Repetición. Un programa imperativo, normalmente realiza su tarea ejecutando repetidamente una secuencia de pasos elementales.

El paradigma imperativo es una abstracción del lenguaje ensamblador. Algunos de los lenguajes de programación que siguen el paradigma imperativo son: Basic, Pascal, Modula y C.

Una clasificación importante del paradigma imperativo es:

Procedural (estructurado). Es un paradigma de programación basado en el concepto de llamado de procedimientos, también conocidos como rutinas, subrutinas, métodos o funciones simplemente contienen series de pasos computacionales, cualquier procedimiento puede ser llamado en cualquier punto durante la ejecución de un programa, incluyendo otros procedimientos o en él mismo. Algunos lenguajes de programación procedimental son: Pascal, C, Basic

Paradigma orientado a objetos. Es un paradigma de programación que define los programas en términos de "clases de objetos", los objetos son entidades que combinan *estado* (es decir, datos) y *comporta-*

miento (esto es, procedimientos o métodos). La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que se comunican entre ellos para realizar tareas. Esto difiere de los lenguajes procedurales tradicionales, en los que los datos y los procedimientos están separados y sin relación, lo que no sucede con los lenguajes orientados a objetos donde los datos y los procedimientos (métodos) forman una entidad llamada *objeto*. Estos métodos están pensados para hacer los programas y módulos más fáciles de escribir, mantener y reutilizar.

Una ecuación para reconocer una aproximación a la Orientación a Objetos es (Jimenez, 2005):

Orientación_Objetos = Clases + Objetos + Herencia + Comunicación con mensajes

Los elementos principales de la programación orientada a objetos son

- Objetos (Sinónimo de instancia). Es la abstracción de alguna cosa en el dominio del problema que refleja la capacidad de un sistema de alcanzar información alrededor de él. Los objetos, por lo tanto, son entidades que tienen atributos (datos) y formas de comportamiento (procedimientos) particulares.

- Clase. Una clase es una descripción de un conjunto de objetos casi idénticos. Una clase consta de métodos y datos que resumen las características comunes de los objetos, incluyendo una descripción de cómo crear un nuevo objeto de la clase. En otras palabras, las clases contienen los anteproyectos para crear objetos.

- Herencia. Es un mecanismo para expresar similaridad entre clases, simplificando definiciones de las clases similares previamente definidas. La herencia permite crear nuevas clases llamadas subclases agregando solamente las diferencias con la clase. En otras palabras la herencia es una partición en subclases más especializadas.

- Comunicación con mensajes. Los objetos tienen la posibilidad de actuar, la acción sucede cuando un objeto recibe un mensaje, que es, una solicitud que pide al objeto que se comporte de alguna forma. Cuando se ejecutan los programas orientados a objetos, los objetos reciben, interpretan y responden a mensajes procedentes de otros objetos. Los mensajes pueden contener información para clasificar una solicitud. Los procedimientos residen en el objeto y determinan cómo actúa el objeto cuando recibe un mensaje. De hecho, los métodos proporcionan el único mecanismo para cambiar los valores de las variables del objeto. (A este hecho se le conoce como encapsulamiento). Los mensajes que reciben el objeto son los únicos conductos que conectan al objeto con el mundo exterior. Estas características de los objetos confieren a la orientación a objetos su ventaja: la orientación a objetos fomenta la modularidad haciendo muy claras las fronteras entre objetos, explícita la comunicación entre los mismo y oculta los detalles de la realización.

A este paradigma pertenecen los lenguajes de programación: SmallTalk, Java, C++.

Paradigmas orientados a eventos. Es un paradigma de programación que se centra en la definición de acciones sobre objetos predefinidos, denominadas ***eventos***. Son eventos típicos: el click sobre un objeto de tipo botón, el hacer doble click sobre el nombre de un archivo para abrirlo, el arrastrar un icono, el pulsar una tecla o combinación de teclas, el elegir una opción de un menú, el escribir en una caja de texto, o simplemente mover el ratón entre otros.

A este paradigma pertenecen los lenguajes de programación: Visual Basic, C Builder, Kylix, Power Builder.

1.2.2 Paradigma Declarativo

El paradigma declarativo señala las características que debe tener la solución, sin describir cómo procesarla. Es decir, especifica qué se desea obtener pero no cómo obtenerlo. Un programa declarativo se construye señalando hechos, reglas, restricciones, ecuaciones,

transformaciones y otras propiedades derivadas del conjunto de valores que configuran la solución. La programación en un lenguaje declarativo es potencialmente de más alto nivel que la programación imperativa.

La característica fundamental del paradigma declarativo es que no existe la asignación ni el cambio de estado en un programa. Las variables son identificadores de valores que no cambian en toda la evaluación. Sólo existen valores y expresiones matemáticas que devuelven nuevos valores a partir de los declarados. A partir de esta información, el sistema debe de ser capaz de derivar un esquema de evaluación que permita computar una solución, es decir, no existe una descripción paso a paso de cómo llegar a la solución.

En términos generales, al desarrollar software en el paradigma de programación declarativo, en contraste con el Imperativo, se tiende a enfatizar más la evaluación de expresiones que la ejecución secuencial de comandos.

Las soluciones declarativas son más cercanas a la manera en que funciona la mente humana al permitir a los programadores describirlas como sistema de expresiones que serán evaluadas. Por otro lado, las soluciones imperativas, al forzar la formulación de algoritmos como una serie de pasos que se realizarán de manera secuencial, son semejantes a la manera en que funciona el computador.

Una clasificación importante del paradigma declarativo es:

Paradigma Lógico. La programación lógica está basada en el cálculo de predicados, incluyendo instrucciones escritas en formas conocidas como cláusulas de Horn. El cálculo de predicados proporciona axiomas y reglas de modo que se puede deducir nuevos hechos a partir de otros ya conocidos. Una cláusula de Horn permite que sólo un nuevo hecho sea deducido en cualquier instrucción simple. Un sistema de cláusulas de Horn permite un método particularmente mecánico de demostración llamado *resolución* (Appleby y Vandekopple, 1998).

Un programa basado en lógica se compone de una serie de axiomas o hechos, reglas de inferencia y un teorema o cuestión por demostrarse. La salida es verdadera si los hechos soportan o apoyan la cuestión, y es falsa en el caso contrario.

A este paradigma pertenecen los lenguajes de programación: Prolog y CLP.

Paradigma Funcional. El objetivo del paradigma funcional es conseguir lenguajes expresivos y matemáticamente elegantes, en los que no sea necesario bajar al nivel de la máquina para describir el proceso llevado a cabo por el programa, evitando el concepto de estado del cómputo. La secuencia de computaciones llevadas a cabo por el programa se rige única y exclusivamente por la reescritura de definiciones más amplias a otras cada vez más concretas y definidas, usando lo que se denominan *definiciones dirigidas* (Guzman, 2000).

Una *definición dirigida* es un modelo matemático de composición funcional donde el resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produce el valor deseado.

En el paradigma Funcional no existe el concepto de celda de memoria que es asignada o modificada; más bien, existen valores intermedios que son el resultado de cálculos anteriores y las entradas a cálculos subsiguientes, ni tampoco existen sentencias imperativas. La programación funcional incorpora el concepto de función como objeto de primera clase, lo que significa que las funciones se pueden tratar como datos (pueden pasar como parámetros, calculadas y devueltas como valores normales, y mezcladas en el cálculo con otras formas de datos).

En este paradigma se concibe la solución como una composición de funciones. Por ejemplo, para ordenar una lista, se puede diseñar la solución como una concatenación de listas más pequeñas, cada una de las cuales ya está clasificada. Esto reduce el problema a seleccionar las listas más pequeñas.

A este paradigma pertenecen los lenguajes de programación: Scheme, Commonlisp.

Paradigmas del lenguaje de base de datos. Las propiedades que distinguen a los lenguajes diseñados para tratar con bases de datos son la persistencia y la administración de cambios. Las entidades de bases de datos no desaparecen después de que finaliza un programa, sino que permanecen activas durante tiempo indefinido como fueron estructuradas originalmente. Puesto que la base de datos, una vez organizada, es permanente, estos lenguajes también deben soportar los cambios. Los datos pueden cambiar y así también pueden hacerlo las relaciones entre objetos o entidades de datos (Appleby y Vandekopple, 1998).

Un sistema de administración de base de datos incluye un l*enguaje de definición de datos* (DDL) para describir una nueva colección de hechos, o datos, y un l*enguaje de manipulación de datos* (DML) para la interacción con las bases de datos existentes.

A este paradigma pertenecen los lenguajes de programación: Sequel, Sql.

Paradigma Heurístico. Se puede definir como aquel tipo de programación computacional que aplica para la resolución de problemas reglas de buena lógica (reglas del pulgar) denominadas heurísticas, las cuales proporcionan entre varios cursos de acción uno que presenta visos de ser el más prometedor, pero no garantiza necesariamente el curso de acción más efectivo (Barzanallana, 2005).

Una heurística es la conclusión del razonamiento humano en un dominio específico. Por ello, este tipo de programación se aplica con mayor intensidad en el campo de la Inteligencia Artificial (IA), y en especial, en el de la Ingeniería del Conocimiento, dado que el ser humano opera la mayoría de las veces utilizando heurísticas.

La Programación Heurística implica la aplicación del conocimiento específico integral de la naturaleza del problema a resolver, que

comprende, además de su operatoria, la manera cómo se genera y que efectos produce. Es una forma de modelar el problema en lo que respecta a la representación de su estructura, estrategias de búsqueda y métodos de resolución. Esto permite minimizar el tiempo de búsqueda de las alternativas más eficaces.

Este paradigma de programación se presenta y utiliza desde diferentes puntos de vista:

Como técnica de búsqueda para la obtención de metas en problemas no algorítmicos, o con algoritmos que generan explosión combinatoria, por ejemplo juegos como damas y ajedrez.

Como un método aproximado de resolución de problemas utilizando funciones de evaluación de tipo heurístico por ejemplo algoritmos A* y AO*

Como método de poda para estrategias de programas que juegan, aunque estos métodos no son realmente heurísticos por ejemplo poda alfa-beta.

La Programación Heurística no ha producido un lenguaje especifico de programación, debido a que las heurísticas, al *ser reglas de sentido común*, se pueden implementar con cualquiera de los lenguajes descritos en los diferentes paradigmas de programación. Por otra parte, al aplicarse este tipo de programación, fundamentalmente, en el campo de la I.A., las heurísticas están siendo implementadas mayoritariamente en herramientas de esta área.

A este paradigma pertenecen los lenguajes de programación: Logo, Prolog, Kee entre otros.

1.3 PARADIGMA FUNCIONAL

1.3.1 Historia

Está basado completamente en un enfoque funcional, en la creación y uso de funciones matemáticas. Este modelo computacional estableció las bases teóricas de lo que más tarde serían los lenguajes de programación funcional.

La popularidad del paradigma funcional se debe en gran medida al éxito del lenguaje de programación Lisp y de sus dialectos, como Scheme. Tanto el Lisp como Scheme son lenguajes que se están utilizando en la actualidad.

Los orígenes teóricos del modelo funcional se remontan al año 1934, cuando Alonso Church introdujo un modelo matemático de computación llamado l*ambda cálculo*.

El Cálculo Lambda es una forma simple de modelar los aspectos computacionales de las funciones. Su estudio ayuda a comprender los elementos de la programación funcional y de la semántica que subyace en los lenguajes, independientemente de los detalles sintácticos de un lenguaje particular de programación funcional. Su nombre proviene de la letra griega "λ" (Lambda) que se utiliza en la notación (Seron, Magallón y Baldassarri, 2006).

A pesar de que en 1934 las computadoras aun no existían, el cálculo lambda se puede considerar como el primer lenguaje funcional de la historia y sus fundamentos fueron la base de toda la teoría de la programación funcional y de los lenguajes funcionales desarrollados posteriormente. Se puede decir que los lenguajes funcionales modernos son versiones de lambda cálculo con numerosas ayudas sintácticas.

1.3.2 Programación funcional

El uso de funciones, con comportamiento análogo al de las funciones matemáticas, para crear programas es lo que se denomina "programación funcional". Un ingrediente esencial de la programación funcional es que el valor de una función queda determinado exclusivamente por el valor de sus argumentos. Por lo tanto cualquier llamada a la función con los mismos argumentos producirá siempre el mismo valor. En programación funcional las funciones deben cumplir con la propiedad de transparencia referencial.

El objetivo del diseño de la programación funcional es intentar copiar el funcionamiento de las funciones matemáticas de la manera más extensa posible. Los programas escritos en un lenguaje funcional están constituidos únicamente por definiciones de funciones, entendiendo éstas no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la transparencia referencial (el significado de una expresión depende únicamente del significado de sus subexpresiones), y por tanto, la carencia total de efectos laterales.

1.3.3 Lenguajes Funcionales

Los matemáticos han resuelto problemas usando el concepto de función, la cual convierte ciertos datos en resultados. Si se conoce cómo evaluar una función usando la computadora, se pueden resolver automáticamente muchos problemas. Este análisis fue realizado por algunos matemáticos que consideraron la computadora una herramienta importante para su trabajo, creando los lenguajes de programación funcionales, los que aprovechan la posibilidad que tienen las funciones para manipular datos simbólicos, y no solamente numéricos, y la propiedad que les permite componer, creando de esta manera, la oportunidad para resolver problemas complejos a partir de las soluciones a otros más sencillos. También se incluyó la posibilidad de definir funciones recursivamente (Steele y Sussman, 1979).

Un lenguaje funcional ofrece conceptos que son muy entendibles y relativamente fáciles de manejar. El lenguaje funcional más antiguo, y seguramente el más popular hasta la fecha, es LISP, diseñado por McCarthy en la segunda mitad de los años 50. Su área de aplicación es principalmente la Inteligencia Artificial. En la década de los 80 hubo una nueva ola de interés por los lenguajes funcionales, añadiendo la tipificación y algunos conceptos modernos de modularización y polimorfismo. (Mcarthy, 1965)

Programar en un lenguaje funcional significa construir funciones a partir de las ya existentes. Por lo tanto es importante conocer y comprender bien las funciones que conforman la base del lenguaje, así como las que ya fueron definidas previamente. De esta manera se pueden ir construyendo aplicaciones cada vez más complejas.

Un lenguaje de programación funcional proporciona:

- Un conjunto de funciones primitivas
- Un conjunto de formas funcionales para construir funciones complejas.
- Una operación de aplicación de las funciones
- Algunas estructuras para almacenar datos

La ejecución de un lenguaje de programación funcional está basada en dos mecanismos fundamentales:

- Asociación de los valores con los nombres (argumentos de las funciones).
- Aplicación de las funciones para calcular nuevos valores

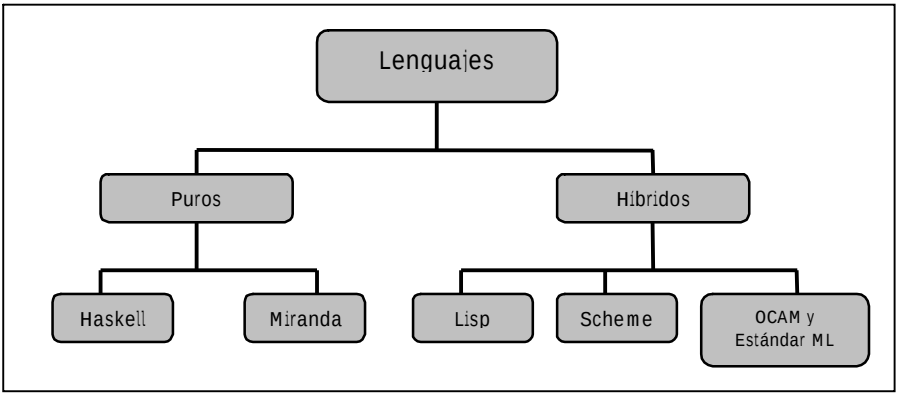
Un lenguaje de programación funcional no necesariamente utiliza variables con el sentido de localización de memoria, ni requiere por lo tanto sentencias de asignación con el significado de los lenguajes imperativos ni la falta de construcciones estructuradas como la secuencia o la iteración (lo que obliga en la práctica a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas), lo que permite liberar al programador del control de las celdas de memoria.

Existen dos grandes categorías de lenguajes funcionales: los funcionales puros y los híbridos. La diferencia entre ambos estriba en que los lenguajes funcionales híbridos son menos dogmáticos que los puros, al admitir conceptos tomados de los lenguajes procedimentales, como las secuencias de instrucciones o la asignación de variables.

En contraste, los lenguajes funcionales puros tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial, algo que no se cumple siempre con un lenguaje funcional híbrido.

Entre los lenguajes funcionales puros, cabe destacar a Haskell y Miranda. Los lenguajes funcionales híbridos más conocidos son: Lisp, Scheme, Ocaml y Standard ML (estos dos últimos, descendientes del lenguaje ML). Ver figura 1.3

Figura 1.3 Categoría de los lenguajes funcionales



Las características fundamentales de los lenguajes funcionales de programación son, según (Guzman, 2000):

El valor de una expresión depende sólo de los valores de sus subexpresiones, si las tiene. La programación funcional pura es una programación sin asignaciones. En realidad, la mayoría de los lenguajes funcionales son impuros, ya que permiten asignaciones. Sin

embargo, su estilo de programación es diferente al de los lenguajes de programación imperativa.

Almacenamiento implícito. El programador no debe preocuparse en manejar el almacenamiento de datos. Una consecuencia de esto es que la implementación del lenguaje debe realizar una "recolección de basura" para recuperar la memoria que se ha usado y no se volverá a utilizar.

Las funciones son valores de primera clase. Una función puede ser el valor de una expresión, pasarse como argumento o colocarse en una estructura de datos. Esto permite potentes operaciones.

CAPÍTULO 2

FUNDAMENTOS DEL LENGUAJE SCHEME

"El éxito
no se logra sólo con cualidades especiales.
Es sobre todo un trabajo de constancia,
de método y de organización"
J.P. Sergent

El lenguaje de programación Scheme es un lenguaje compacto y con un alto grado de abstracción; por eso, resulta adecuado para ser presentado y aprendido en las asignaturas introductorias a la programación, donde el énfasis se hace en la metodología de resolución de problemas utilizando lenguajes de programación. Scheme permite plantear soluciones a requerimientos complejos a través de programas cortos y elegantes, de tal manera que se convierte en un aliado para resolver problemas y no un problema más que resolver.

En este capítulo se presenta los fundamentos del lenguaje de programación Scheme.

2.1 HISTORIA

Los orígenes del lenguaje Scheme se trasladan al año 1975, cuando Gerald Jay Sussman y Guy Lewis Steele Jr., se encontraban estudiando la teoría de los actores, como un modelo de computación desarrollado por Carl Hewitt en el Instituto Tecnológico de Massachussets (MIT). De

acuerdo con esta teoría, cada objeto es una entidad computacionalmente activa capaz de recibir y reaccionar a los mensajes. A estos objetos y a los mensajes intercambiados por ellos, Hewitt, los llamó actores.

Cuando Sussman y Steele afrontaron dificultades para entender algunos aspectos teóricos y prácticos relacionados con el trabajo de Hewitt, decidieron desarrollar un pequeño intérprete de este lenguaje usando MacLisp. Esta decisión les permitió crear actores con la misma facilidad que se crean las funciones en Lisp (y usando casi los mismos mecanismos). De tal manera que, el paso de mensajes se pudo expresar sintácticamente en la misma forma que se invoca una función. La única diferencia fue que una función regresaba un valor y un actor no regresaba nada, más bien invocaba una continuación, o sea, otro actor que sabía de su existencia.

Sussman y Steele se sintieron muy satisfechos con su mini-interprete y decidieron llamarlo *Schemer*, sin embargo el sistema operativo que manejaban en ese momento, el *Sistema Operativo de Tiempo Compartido* (ITS) desarrollado por el MIT a finales de los años 60, limitaba los nombres de sus ficheros a seis letras, por lo cual fue necesario reducirlo a Scheme, que es el nombre con el que se conoce hoy en día.

Posteriormente, en 1976, los autores de Scheme publicaron dos artículos (Steele, 1976), en los que se hablaba sobre la semántica de este lenguaje y se enfatizaba que éste intérprete soportaba eficientemente los principales paradigmas de programación de ese momento, a saber: imperativo, funcional y orientado a objetos. La última publicación se convirtió de hecho en la propuesta de tesis de maestría de Steele, que culminó en el desarrollo de un compilador de Scheme llamado Rabbit (Steele, 1978).

Finalmente, se puede concluir que hoy en día Scheme es un lenguaje simple, pero poderoso; pequeño, pero flexible. Su naturaleza lo hace ideal para la enseñanza, por su notable facilidad para incorporar diversos paradigmas con sólo unas pocas primitivas (Coello, 1996).

2.2 CARACTERÍSTICAS DEL LENGUAJE SCHEME

A pesar que Scheme es un lenguaje muy pequeño y simple, es asimismo sumamente poderoso, a tal grado que es utilizado como herramienta para el desarrollo de aplicaciones y software de sistemas (Coello, 1996) (Lucena, 2003). Las siguientes son algunas de sus características:

Transparencia referencial. Las variables son identificadores que referencian datos o funciones y no lugares predeterminados de almacenamiento.

Funciones de orden superior. Este tipo de funciones devuelven otras funciones o aceptan funciones como parámetros. Permiten construir funciones complejas a partir de funciones simples.

Evaluación perezosa. Se evalúan los parámetros de una función únicamente cuando se necesitan. Esta característica permite manejar estructuras de datos de tamaño infinito.

Abstracción de datos. Construcción de datos complejos a partir de datos más simples.

Gestión automática de memoria. Scheme identifica los objetos que no serán utilizados en el futuro, es decir, aquellos cuyas referencias han sido canceladas, y libera el espacio de memoria para poder ocuparlo con otros objetos. A este proceso se le llama *recolección de basura*. Esto tiende a simplificar el código fuente de los programas ya que libera al programador de la tarea de llevar registro de la memoria utilizada y de implementar código para liberarla.

Sintaxis muy simple y regular. La sintaxis de Scheme es muy reducida, comparada con otros lenguajes, emplea una notación prefija por lo que no requiere reglas de precedencia ni operadores para todas las llamadas a funciones. Utiliza paréntesis sucesivos para definir funciones y espacios para separar sus parámetros.

Proporciona poderosas representaciones numéricas. Scheme cuenta con una rica variedad de formatos y representaciones numéricas que incluye: números enteros, de punto fijo y flotante, complejos y racionales; si bien sólo unas cuantas de ellas son requeridas por el estándar de lenguaje.

Es un lenguaje orientado a las expresiones. Scheme no hace diferencia entre funciones y datos, todo lo que maneja son expresiones. Por ejemplo: (define a 3) es una referencia a un dato en tanto que (define a (read)) es una referencia al resultado de una función y ambas son expresiones del lenguaje.

Es un lenguaje con tipos dinámicos (No Tipado). Esto significa que no tienen que declararse las variables que se van a utilizar en un programa, porque el tipo es una propiedad de los objetos y no una propiedad de las variables que nombran a los objetos. Los sistemas de tipos dinámicos pueden considerarse como una forma extremadamente simple y general de polimorfismo (Clinger, 1988). La mayor desventaja de los lenguajes que cuentan con tales sistemas es que resulta imposible capturar los errores de tipos en tiempo de compilación.

Paso de parámetros por valor. Scheme pasa sus argumentos por valor, i.e. al invocar una función se pasa el valor del parámetro real y no la referencia como sucede en otros lenguajes.

2.3 ESTRUCTURA DE UN PROGRAMA EN SCHEME

Un programa diseñado con lenguaje Scheme no tiene una estructura impuesta y no hay ninguna función principal. Se puede definir un bloque de funciones como principal y a partir de él se invoca otras funciones del lenguaje o definidas por el usuario.

En el cuadro 2.1 se propone una estructura básica de un programa en Scheme con tres elementos: el primero corresponde a un comentario con información general del programa, como puede ser: autor, fecha, descripción; el segundo comprende las funciones definidas por el

usuario, estas deben estar definidas antes de ser invocadas, por ello conviene escribirlas antes del bloque principal; y finalmente, el bloque principal, éste será el primero en ejecutarse, el cual debe iniciarse utilizando la función *begin*, de manera que desde allí se organiza la ejecución de las demás funciones. En programas pequeños que no incluyan funciones definidas por el usuario todas las tareas se desarrollarán en el bloque principal.

Cuadro 2.1. Estructura básica de un programa en Scheme

```
    #| Información del programa |#
```

```
;  Función definida por el usuario
(define mi-funcion
       (lambda (parámetros)
              cuerpo de la función
       )
)
```

```
; Bloque principal
(begin
       ;llamado a funciones
       (mi-funcion)
)
```

En el cuadro 2.2 se presenta un programa sencillo en Scheme que consta de un comentario y el bloque principal. Este programa contiene las funciones necesarias para mostrar un mensaje en pantalla.

Cuadro 2.2. Ejemplo de un programa en Scheme

```
1  ; Mi primer programa en Écheme
2  (begin
3      (display "Hola mundo!")
4  )
```

```
Welcome to DrScheme, version 360.
Language: Standard (R5RS).
Hola Mundo!
>
```

Como se observa en el cuadro 3.2, la parte sombreada corresponde al código fuente del ejemplo y la parte no sombreada, a la información de salida del programa. Esta es la notación que se utiliza en todos los ejemplos del libro.

2.4 ELEMENTOS DEL LENGUAJE SCHEME

2.4.1 Comentarios

Son las líneas de texto que el programador escribe para brindar orientaciones o explicaciones relacionadas con el código del programa, sin que afecten la ejecución del mismo, es decir que el compilador no las tiene en cuenta. Se utilizan dos tipos de comentarios: de línea y multilínea.

Los comentarios de línea comienzan con punto y coma (;). Estos pueden ocupar una línea o pueden escribirse después de una instrucción, como se aprecia en las líneas 3, 4 y 6 del Cuadro 2.3. Los comentarios multilínea se escriben entre los símbolos: #| y |#, como se indica en las líneas 1 y 2.

Cuadro 2.3. Uso de comentarios

```
 1   #| En este programa se muestra …
 2       El uso de comentarios |#
 3   (begin       ; inicio del programa
 4     ; La siguiente línea imprime el texto "Hola, mundo!"
 5      (display "Hola, mundo!");
 6   )               ; fin del programa
```

```
Hola Mundo!
>
```

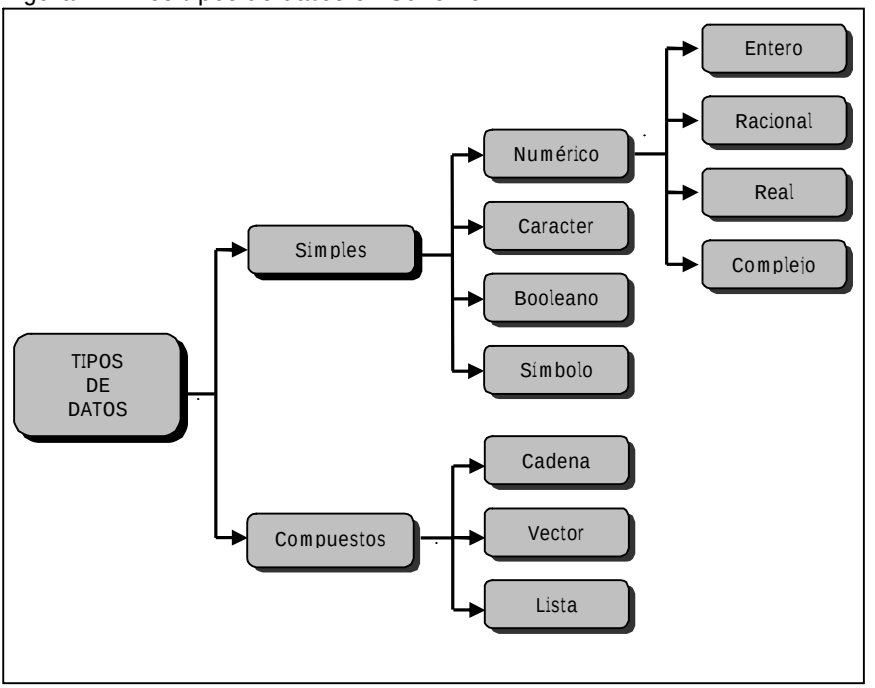
2.4.2 Tipos de datos

Scheme es un lenguaje de programación no tipado, de esta manera sus variables guardan cualquier valor sin importar si es numérico,

alfanumérico, booleano o de cualquier otro tipo. De esta manera, en un espacio de memoria se podría guardar un nombre, un número telefónico o un tipo de dato más complejo como un vector o un objeto.

Por lo anterior, en Scheme es importante distinguir claramente los diferentes tipos de datos, porque la clase de información almacenada en sus variables no es responsabilidad del compilador sino del programador. En la figura 2.1 se ilustra los tipos de datos que soporta Scheme y como están organizados.

Figura 2.1. Los tipos de datos en Scheme



Tipos de datos simples. Son aquellos que representan datos primarios o atómicos. En Scheme se tiene cuatro tipos de datos simples: números, caracteres, booleanos y símbolos.

Numéricos. En este grupo se asocian todos los tipos de datos con los cuales se pueden realizar operaciones matemáticas, pueden ser enteros o tener coma flotante, y pueden estar en diferentes bases: binario, octal, decimal o hexadecimal. Para indicar a qué sistema numérico pertenece la variable se utiliza un prefijo en su definición, así: #b para binarios, #o para octales, #d para decimales (tipo por defecto), #x para hexadecimales, por ejemplo: *#d*3.14, -8, *#x*3AF8, *#b*0100, *#o*543.

A su vez, los datos de tipo numérico, se clasifican en:

- Enteros. Representan los números sin componente fraccionario (exactos) y pueden ser tanto positivos o negativos, por ejemplo: 20, -20, 1450, -1450, 8, -8.

- Racionales. Representan todos los números de la forma *a/b* donde *a* y *b* son enteros y *b* es diferente de 0, por ejemplo: 3/4, -4/8, 5/8, -2/3.

- Reales. Representan todos los números que poseen componente fraccionaria, por ejemplo: 12.5, -12.5, 278.004, -51.23.

- Complejos. Representan todos los números que poseen una parte real y una parte imaginaria (i), por ejemplo: 2+3i, -154.3+87i, 10+24i.

Si una variable tiene un valor numérico puede pertenecer a más de un tipo, por ejemplo si el contenido es 5 entonces podría ser interpretada como numérica, entera, racional, real y compleja.

Caracter. Este tipo comprende cada uno de los elementos de la tabla de codificación ASCII (American Standard Code for Information Interchange). Un caracter puede ser una letra, un digito o un caracter especial. En Scheme, para indicarle al compilador que un dato es de tipo caracter, se utiliza el prefijo: #\. Ejemplos: #\a, (carácter `a´), #\B (carácter `B´), #\p (carácter `p´), #/newline, (cambio de línea) #/space (espacio en blanco).

Booleano. Este tipo de dato sólo puede tomar uno de dos valores: *verdadero*, representado por el símbolo #t o *falso*, representado por el símbolo #f. Cualquier valor distinto de #f, incluyendo la lista vacía, se interpreta como *verdadero*.

Símbolo. Es el tipo de dato nativo o neutro que utiliza Scheme para asociar cualquier valor alfanumérico que no pertenece a ninguno de los tipos de datos anteriormente listados. Cualquier conjunto de caracteres alfanumérico que no indique su tipo será asociado a este tipo.

La única comparación que puede efectuarse con datos de tipo *symbol* es la igualdad. El resultado es verdadero si y solamente si los dos símbolos son idénticos:

Ejemplos: (symbol=? Hola Hola) **=>** #t
(symbol=? Hola hola) **=>** #f

Tipos de datos compuestos. Son colecciones de datos que se generan por la agrupación de datos simples o compuestos, por ejemplo: una cadena, una lista o un vector.

Cadena (string). Es una sucesión de caracteres escrita entre comillas dobles (" cadena "), puede contener números, letras y símbolos. Ejemplo: *"San Juan de Pasto"*, *"DrScheme, version 360."*, *"Programación Funcional"* .

Vectores o Arreglos. Son estructuras de longitud predefinida que permiten agrupar elementos identificándolos con un índice que corresponde a la posición que ocupa de la colección. El acceso a los elementos de un arreglo se realiza mediante las funciones que el lenguaje ofrece y que se estudian en el capítulo dedicado a este tema. Ejemplo, para guardar cinco marcas de automóviles se puede definir el vector *marcas* y en cada posición guardar una de ellas, gráficamente el vector se vería como en la figura 2.2.

Figura 2.2. Ejemplo de vector

marcas	Ford	Mazda	Renault	Honda	Chevrolet	Datos
	0	1	2	3	4	Índice

Lista. Es un conjunto de elementos de igual o diferente tipo de dato, que se encuentran agrupados e identificados con un mismo nombre; se diferencian de los arreglos en que no tiene una dimensión preestablecida sino que se puede construir de forma dinámica en tiempo de ejecución. Para definir una lista se escriben los elementos entre paréntesis () y precedidos por una comilla sencilla..

Ejemplos:

- Lista de vocales: '(a e i o u)
- Lista de edades: '(30 55 18 12)
- Lista de nombres: '("Juan" "Pedro" "Hugo")
- Lista de diferentes tipos de datos: '(8 "Ruisnar" 2007)

Funciones para tipos de datos. En Scheme, una variable puede hacer referencia a un dato de cualquier tipo ya que se trata de un lenguaje no tipado. No obstante, para realizar operaciones con los datos es necesario conocer su tipo, ya que algunas operaciones no están permitidas sobre ciertos tipos, por ejemplo, las operaciones aritméticas solo pueden efectuarse sobre operandos numéricos, de igual manera que no se puede convertir a mayúsculas o minúsculas los dígitos y caracteres especiales, ya que esta es una operación exclusiva para caracteres del alfabeto.

Para reconocer el tipo al que pertenece un dato Scheme ofrece el conjunto de funciones que se presenta en el cuadro 2.4.

Cuadro 2.4 Funciones tipos de datos

Función	Descripción (Determina ...)
(bolean? var)	Si es un booleano
(number? var)	Si es un número
(integer? var)	Si es un número entero
(racional? var)	Si es un número racional
(real? var)	Si es un número real
(complex? var)	Si es un número complejo
(char? var)	Si es carácter
(symbol? var)	Si es símbolo
(string? var)	Si es cadena
(vector? var)	Si es vector
(list? var)	Si es una lista

Estas funciones devuelven un dato booleano, verdadero (#t) si el contenido de la variable corresponde al tipo que indica la función y falso (#f) en caso contrario.

Cuadro 2.5. Programa para evaluar tipos de datos

```
 1   ; Programa para evalar el tipos de datos
 2   (begin                              ;inicio del programa
 3     (display "digite un dato: ")     ; Imprime un mensaje
 4     (define x (read))     ; Captura un dato para ser
     evaluado
 5     ;La siguientes instrucciones evalúan el tipo de dato
     digitado
 6     (if (symbol? x)   (display "Es Símbolo, "))
 7     (if (char? x)     (display "Es Caracter, "))
 8     (if (string? x)   (display "ES Cadena de texto, "))
 9     (if (real? x)     (display "Es Real, "))
10     (if (number? x)   (display "Es Número, "))
11     (if (integer? x)  (display "Es Entero, "))
12     (if (rational? x) (display "Es Racional, "))
13     (if (complex? x)  (display "Es Complejo, "))
14     (if (vector? x)   (display "Es un Vector, "))
15     (if (list? X)     (display "Es una Lista, "))
16     )  ;fin del programa
```

```
Digite un dato: X3AF8
Es Símbolo
```

En el programa que se presenta en el cuadro 2.5 se lee un dato cualquiera y utilizando las funciones presentadas en el cuadro 3.2 se determina a qué tipo pertenece.

2.4.3 Variables

Una variable es una referencia a un dato que reside en alguna posición de memoria. Está vinculada a una tupla de atributos: nombre, dirección, tipo y valor (Appleby y Vandekopple).

El nombre de una variable es una cadena que puede estar conformada por caracteres alfabéticos, dígitos y el caracter guión (-), no debe contener espacios en blanco ni otros caracteres especiales que puedan tener otra función en el programa, como los operadores.

Todo dato asociado a una variable ocupa espacio en memoria y la variable está ligada a la primera posición ocupada por el dato; no obstante, en Scheme, el programador no necesita encargarse del manejo de las direcciones de memoria ya que el lenguaje lo hace de forma automática.

Scheme cuenta con diversos tipos de datos (ver sección 3.4.2), pero por tratarse de un lenguaje no tipado, el tipo no está ligado al nombre ni a la dirección de memoria, sino a los valores o datos, de manera que una variable puede estar asociada a cualquier valor, indiferente del tipo. El tipo asociado al valor se establece en la definición o digitación del dato mediante los prefijos explicados en la sección 2.4.2, o se toma el que corresponda por omisión.

Definición de variables

Un programa necesita mantener una lista de las variables que utiliza, de manera que cuando sus nombres aparezcan en una expresión éste pueda verificar su existencia y buscar su valor. La definición de una variable es la notificación formal de su existencia.

Para definir una variable se utiliza la función *define*, cuya sintaxis es:

```
(define nombre valor)
```

Donde *nombre* es el identificador que permitirá hacer uso de la variable y *valor* es el dato asociado. Cuando se desea definir una variable sin asociar ningún valor se utiliza la palabra reservada *null*.

Ejemplos:

```
(define numero 10)
```

Define la variable *numero* y la liga con el valor *10*.

```
(define nombre "Gerald Jay Sussman")
```

Define la variable *nombre* y la asocia con una cadena de texto.

```
(define numero-binario #b101)
```

Define la variable *numero-binario* y la liga con el valor *101* indicando que es un número binario.

```
(define letra #\a)
```

Define la variable *letra* y la asocia con la letra *a* indicando que es un tipo caracter.

```
(define numero-estudiantes null)
```

Define la variable *numero-estudiantes* sin hacer referencia a ningún valor.

Una variable puede ser definida en cualquier parte del programa, excepto dentro de bloques condicionales o iterativos. El lugar donde se define determinará el ámbito en el que la variable es reconocida.

Asociación de valores a variables

Una variable puede hacer referencia a diferentes valores durante la ejecución de un programa. La primera ligadura *variable – valor* se establece en el momento de la definición de la variable mediante la función *define*, después, para hacer que la variable se asocie con otro valor es necesario utilizar la función *set!*. La sintaxis de la función *set!* es:

```
(set! variable  valor)
```

La función *set!* solo puede utilizarse con variables previamente definidas. Como ejemplo se asocia nuevos valores a las variables definidas anteriormente.

```
(set! numero 15)

(set! nombre "Guy Lewis Steele Jr")

(set! numero-binario #b110001)

(set! letra #\c)

(set! numero-estudiantes 25)
```

Ámbito de las variables

El ámbito de una variable es el espacio en el que ésta es reconocida y puede ser utilizada para construir expresiones o para hacer referencia a su resultado. El ámbito puede estar dado por un bloque de expresiones como un bloque condicional o iterativo, una función, o la totalidad del programa.

Según el ámbito las variables se clasifican en: globales y locales. Las variables globales son conocidas en todo los módulos o funciones del programa, para ello se definen al inicio del código, fuera de cualquier función o bloque indicado con *begin*. Las variables locales, por su parte, se definen al interior de una función o bloque y su existencia se limita a dicho espacio. En el cuadro 2.6 se muestra el manejo de variables, indicando su ámbito.

Cuadro 2.6. Ejemplo de uso de variables

```
1.   ;  este  programa  calcula  la  sumatoria  y  la
    diferencia de dos numeros
2.
3.   (define suma null)        ; variable global
4.   (define diferencia null)  ; variable global
5.
6.   (define sumar-restar
7.     (lambda (x y)
8.       (set! Suma (+ x y))
9.       (set! diferencia (- x y))
10.      )
11.    )
12.
13.  (begin
14.    (define a 12)          ; variable local
15.    (define b 15)          ; variable local
16.    (sumar-restar a b)
17.    (newline)
18.    (display "La suma es: ")
19.    (display suma)
20.    (newline)
21.    (display "La diferencia es: ")
22.    (display diferencia)
23.    )
```

```
La suma es: 27
La diferencia es: -3
```

En este programa se aprecia la definición de variables en las líneas 3, 4, 14 y 15. Las dos primeras son variables globales y por ello pueden ser utilizadas tanto en el bloque principal *begin* como en la función *sumar-restar*. Las definidas en las líneas 14 y 15 son locales y solo existen dentro del bloque *begin*. Las variables *x* e *y* que se utilizan como parámetros de la función *sumar-restar* son locales a esta función.

2.4.5 Expresiones

Una expresión es una combinación de operadores y operandos que genera algún resultado. Estas construcciones son muy importantes en

el desarrollo de programas, ya que a través de ellas se indica las operaciones ha ejecutar (Chaves, 2004).

Considerando los tipos de operadores se tienen, también, tres tipos de expresiones: aritméticas, relacionales y lógicas. Las primeras se construyen con datos numéricos y operadores aritméticos, su resultado es también un dato numérico, algunos ejemplos son: sumar dos números, calcular el área de una figura geométrica, resolver una ecuación lineal; las relacionales se forman con operadores que indican comparación como: igualdad, mayor que, menor que, diferente, y los operandos pueden ser numéricos o no numéricos, pero todos los que conforman la expresión deben ser del mismo tipo, el resultado es un dato de tipo lógico: verdadero o falso; las lógicas se componen con operandos y operadores lógicos y son básicamente tres: la conjunción (y), la disyunción (o) y la negación, su resultado es también un dato lógico.

La forma como se combinan los operadores y los operandos se conoce como notación. Existen básicamente tres tipos de notaciones: Prefija, Infija y Postfija. En la notación prefija se escribe el operador seguido de los operandos, en la infija se escribe un operando, un operador y otro operando y así sucesivamente se pueden combinar más de un operador y sus datos; en la postfija se presentan primero los operandos y al final el operador. En el cuadro 2.7 se presenta la expresión para calcular el perímetro de un rectángulo, utilizando las tres notaciones.

Cuadro 2.7 Ejemplos de Notaciones

Expresión	Notación	Variables
(+ (* 2 a) (* 2 b))	Prefija	a: largo b: ancho
2 * a + 2 * b	Infija	
((2 a *) (2 b *) +)	Postfija	

Scheme, por ser un lenguaje funcional, los operadores son funciones y los operandos son sus parámetros; por esto, las expresiones se construyen con notación prefija.

En Scheme cada expresión comienza con un paréntesis izquierdo, luego el operador, a continuación los operandos y termina con el paréntesis derecho. Tanto el operador como los operandos se separan por un espacio. Si una expresión tiene subexpresiones es necesario que éstas tengan sus propios paréntesis. En el cuadro 2.8 se presentan ejemplos de expresiones. *a*, *b*, *c* y d son variables numéricas.

Cuadro 2.8 Ejemplos de expresiones

(+)	Expresión sin argumentos. El resultado es 0
(+ *a*)	Expresión con un solo argumento. El resultado es *a*
(+ *a b c d*)	Se suman los cuatro valores
(- *a b c d*)	Al primer valor se le restan los siguientes
(+ (* *a b*) (/ *c d*))	Se multiplican *a* y *b*, se divide *c* sobre *d* y los resultados de estas dos expresiones se suman

2.4.6 Funciones

Una estrategia muy utilizada para resolver problemas muy complejos o que requieren programas con muchas líneas de código es conocida como divide y vencerás. Esta consiste en descomponer un problema en cierto número de problemas más pequeño, resolver sucesiva e independientemente todos estos problemas y luego combinar las soluciones obtenidas para obtener la solución al problema principal. (Brassard y Bratley, 1997).

Las subsoluciones que resultan de esta descomposición se denominan funciones y procedimientos. Una función es un subprograma que recibe uno o más parámetros, ejecuta una tarea específica y devuelve un único valor como resultado, al programa o función que la invocó; mientras que un procedimiento desarrolla una o más tareas y no genera ningún valor de retorno.
La implementación de programas haciendo uso de procedimientos y funciones tienen entre otras ventajas: facilitar la solución del problema ya que el programador se concentra únicamente en una pequeña parte de éste, escribir programas más legibles, más fáciles de corregir y mantener, y por otra parte, las funciones pueden agruparse

conformando librerías que se pueden utilizar en más de un programa. Las funciones se clasifican en dos tipos: internas y externas.

Funciones internas. Son aquellas que se han definido en el lenguaje de programación, se distribuyen en forma de librerías o de APIs y se utilizan como base para el desarrollo de los programas. Estas atienden necesidades generales y suelen agruparse dependiendo del dominio en que pueden ser utilizadas, por ejemplo: matemáticas, cadenas, fechas, archivos. Scheme cuenta con un interesante conjunto de funciones internas, el cuadro 2.9, presenta algunas como ejemplo.

Funciones externas. También se conocen como funciones definidas por el usuario y son todas aquellas escritas por el programador para atender necesidades particulares en sus aplicaciones.

Cuadro 2.9 Ejemplos de funciones internas

(Abs *n*)	Devuelve el valor absoluto de *n*.
(Cos *n*)	Devuelve el coseno de *n* (en radianes).
(Quotient *a b*)	Devuelve el cociente entero de la división de dos números.
(Random *n*)	Devuelve un número seudo aleatorio entre 0 y *n* - 1.
(Remainder *a b*)	Devuelve el residuo de la división entera de *a* sobre *b*
(Round *n*)	Devuelve el número entero más cercano a *n*.
(Sin *n*)	Devuelve el seno de *n* (en radianes).
(Sqrt *n*)	Devuelve la raíz cuadrada de *n*.
(Truncate *n*)	Devuelve la parte entera de *n*

Implementación de funciones en Scheme. En términos generales, una función consta de dos partes: la definición y la implementación. En la definición se le da un identificador con el cual invocarla y se establecen los parámetros con los que ha de trabajar, en la implementación se escriben los cálculos o tareas que ha de ejecutar.

En Scheme una función se define haciendo uso de las palabras ***define*** y ***lambda***, mientras que la implementación puede hacer uso de cualquiera de las funciones disponibles tanto internas como externas. La forma general de escribir una función se muestra en el cuadro 2.10:

Cuadro 2.10 Definición de una función

```
(define identificador
      (lambda (lista-parámetros)
            ; implementación del cuerpo de la función
            (función-1)
            (función-2)
            (función-n)
      ) ;cierra lambda
) ; cierra define
```

Ejemplo: Escribir una función para sumar dos números.

En el cuadro 2.11 se muestra el código de este programa:

Cuadro 2.11 Función para sumar dos números

```
 1    ; función definida por el usuario
 2    (define suma
 3         (lambda (x y)
 4              (+ x y)
 5          )
 6    )
 8    ; programa principal
 9    (begin
10        (define a (read))
11        (define b (read))
12        (display (suma a b))
13     )
```

```
4
3
7
>
```

Después de ejecutarse todas las expresiones que forman parte del cuerpo de la función ésta termina. El último valor que se haya calculado en la ejecución de la función estará asociado a su identificador y es devuelto a la función que la invocó.

En el ejemplo, entre las líneas 2 y 6 se escribe el código de la función *suma*. Obsérvese que después de la palabra ***define*** aparece *suma* que es el identificador de la función, luego se escribe la palabra ***lambda***, que indica que se está definiendo una función, seguida de los parámetros *x* e *y*. En el cuerpo de la función únicamente se ha incluido una expresión para calcular el resultado de la suma: (+ x y).

A partir de la línea 9 se escribe el programa principal, en este se lee los dos números y se los relaciona con las variables *a* y *b*, las mismas que se envían como parámetros al invocar la función en la línea 12: (suma a b). En la parte baja del cuadro se observa la salida del programa si se prueba con los números 4 y 3.

CAPÍTULO 3

ENTRADA Y SALIDA DE DATOS

"La persona que sabe lo que quiere
tiene medio camino andado
para conseguir su objetivo".
Séneca

Un programa es una herramienta lógica para el procesamiento de datos en el que se puede apreciar dos partes importantes: la implementación de los procesos y la interacción con el usuario o interfaz de usuario. Es a través de la interfaz de usuario que el programa recibe los datos, para llevar a cabo los procesos, y devuelve los resultados.

Scheme implementa la entrada y salida de datos a través de puertos. Un puerto puede representar una cadena de caracteres, un archivo o un dispositivo como el teclado o la pantalla (Kelsey, Clinger y Rees, 1998) (Sitaram, 2004). Cuando en una expresión de entrada o salida no se indica el puerto, esta toma el puerto por defecto.

En este capítulo se presenta las funciones de Scheme que permiten la comunicación entre el usuario y el programa a través de los dispositivos básicos: teclado y pantalla. Considerando que antes de leer un dato es necesario mostrar, al usuario, información sobre lo que el programa requiere, se presenta primero las funciones de salida y luego las de entrada.

3.1 ESCRITURA DE DATOS

Todo programa, por pequeño que sea, reporta algún tipo de información para el usuario. La salida de datos puede hacerse a través de la pantalla, la impresora o dirigirse hacia un archivo (Chaves, 2004); el dispositivo por omisión es la pantalla.

Scheme ofrece tres funciones, dos de salida de datos: *write*, *display* y una para saldo de línea: *newline*.

3.1.1 Función *display*

Esta es la función de salida más utilizada. Recibe un argumento y lo envía a un puerto de salida. Su sintaxis es:

```
(display objeto [puerto])
(display objeto)
```

Objeto hace referencia a cualquier variable o dato. Los datos mostrados por *display* pueden ser de tipo simple: un número o un carácter, o compuesto: una cadena, un vector o una lista. El puerto es un argumento opcional, sino no se especifica la salida se hará por pantalla, si se desea otro destino es menester indicarlo.

Ejemplo 3.1. Un programa que utiliza la función display para mostrar datos de diferente tipo.

En el cuadro 3.1 se muestra la impresión de datos de los tipos más utilizados. En las líneas 3, 7, 9, 11 y 13 se imprime cadenas. Estas se escriben entre comillas dobles (" ") y admiten caracteres no imprimibles como **\n** para insertar un salto de línea o **\t** para una tabulación. En la línea 8 se imprime un carácter, para especificar su tipo se utiliza #\; si se omite este indicador se generará un error de variable no definida. Los datos numéricos y las variables se escriben directamente, como en las líneas 12 y 14.

Cuadro 3.1. Ejemplo de la función display.

```
1. #|Programa: prog031
2.   Función : ejemplos de la función display
3. ------------------------------------------------------|#
4. (begin
5.   (define var 10)
6.   (display "Ejemplos de impresión con display")
7.   (display "\n Caracter: \t")
8.   (display #\a)
9.   (display "\n Dato booleano: \t")
10   (display #t)
11   (display "\n Número: \t")
12   (display 3.5)
13   (display "\n Contenido de una variable: ")
14   (display var)
15  )
```

```
Ejemplos de impresión con display
 Caracter:          a
 Dato booleano:     #t
 Numero:            3.5
 Contenido de una variable: 10
```

La función display solo admite un parámetro, esto implica que si se necesita imprimir varios datos es necesario utilizar una expresión por cada uno. O en su defecto, primero agruparlos en un vector o en una lista.

3.1.2 Función *write*

Es una función para imprimir datos en el puerto que se especifique. A diferencia de la función display esta si imprime los caracteres que especifican los tipos de datos. Por ejemplo: si se imprime una cadena esta aparecerá entre comillas, si se imprime un carácter éste estará precedido por #\.

La sintaxis de la función write es:

```
(write objeto [puerto])
(write objeto)
```

Donde objeto se refiere al dato o colección de datos a imprimir y puerto al dispositivo de destino, puede ser la pantalla, la impresora o un archivo. El puerto es un parámetro opcional, si se omite la salida se dirigirá al puerto por defecto que es la pantalla.

Ejemplo 3.2. Un programa que utiliza la función *write* para mostrar datos de diferente tipo.

Cuadro 3.2. Ejemplo de la función *write*

```
1.   #|Programa: prog032
2.     Función : ejemplos de la función write
3.   ------------------------------------------|#
4.   (begin
5.     (define var 10)
6.     (write "Ejemplos de impresión con write")
7.     (newline)
8.     (write " Caracter: ")
9.     (write #\a)
10.    (newline)
11.    (write " Dato booleano: ")
12.    (write #t)
13.    (newline)
14.    (write " Número: ")
15.    (write 3.5)
16.    (newline)
17.    (write " Contenido de una variable: ")
18.    (write var)
19.   )
```

```
"Ejemplos de impresion con write"
" Caracter: "#\a
" Dato booleano: "#t
" Numero: "3.5
" Contenido de una variable: "10
```

El programa del cuadro 3.2 imprime la misma información que el del cuadro 3.1, pero al utilizar la función write y no display la salida será ligeramente diferente. Las cadenas se muestran con las comillas y no permite utilizar los caracteres no imprimibles \n y \t para insertar salto de línea tabulación. El carácter se presenta con el indicador #\ como parte del dato.

Por las diferencias indicadas se afirma que: la función display se utiliza cuando los datos impresos serán leídos por las personas, mientras que la función write, cuando la impresión será leída por el computador (Kelsey, Clinger y Rees, 1998). En consecuencia, en este libro se utiliza como función de salida únicamente *display*. La función *write* se reserva para las impresiones en archivo.

3.1.3 Función *newline*

Esta es una función de salida que no recibe argumento y por tanto no puede imprimir nada. Su funcionalidad se limita a imprimir un salto de línea. Su sintaxis es:

```
(newline [puerto])
(newline)
```

El puerto es un parámetro opcional. Si no se especifica la salida se dirigirá al dispositivo por defecto: la pantalla.

La función *newline* toma relevancia si se tiene en cuenta que la función *display* solo admite un parámetro y por ende cuando se imprime variables y números no se puede incluir los caracteres no imprimibles (\n) que indican salto de línea. Y cuando se utiliza la función *write*, es necesario utilizar *newline* para generar los saltos de línea. En el ejemplo del cuadro 3.2 se puede apreciar la utilización de esta función.

3.2 LECTURA DE DATOS

La lectura consiste en tomar datos desde un medio externo o desde un archivo y llevarlos a la memoria donde pueden ser procesados por el programa (Joyanes, 1988). El dispositivo predeterminado es el teclado.

Scheme cuenta con tres funciones para la lectura de datos: *read*, *read-char* y *read-line*. Cada una de ellas toma un dato desde el puerto de entrada, pero para tener una referencia a éste es necesario utilizarla

en combinación con la función *define* de manera que el valor leído se asocie a una variable, de la forma:

```
(define objeto (read))
(define caracter (read-char))
(define cadena (read-line))
```

En cada una de estas líneas se define una variable (objeto, carácter o cadena) para hacer referencia al dato leído desde el teclado.

3.2.1 Función *read*

Esta función captura el siguiente objeto desde el dispositivo de entrada asociado al puerto. Su sintaxis es:

```
(read [puerto])
(read)
```

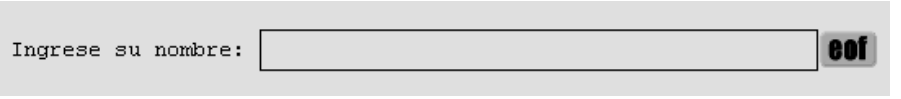
El puerto es un argumento opcional. Si no se escribe la lectura se hará desde el dispositivo predeterminado: el teclado. Por ello, para especificar una lectura por teclado basta con utilizar la función sin argumentos.

La función *read* permite leer cualquier tipo de dato, pues no está asociada a ninguno de ellos. No obstante, por las características del objeto se puede determinar a que tipo pertenece. Si la información entrada no es numérica y no tiene ningún formato que indique su tipo será tratada como el tipo más general en Scheme: Symbol.

Cuado se utiliza la función *read* es necesario que al digitar el dato se incluya los indicadores de formato. Si se quiere que el dato sea tratado como una cadena es necesario escribirlo entre comillas, si es un carácter hay que anteponerle #\, y de igual manera para los otros tipos. (Para mayor información sobre los indicadores de tipos de datos consultar la sección 2.4.2).

Al ejecutarse la función *read* se presenta un cuadro de entrada como el que se muestra en la figura 3.1. Para confirmar la entrada puede hacer clic en el botón *eof* o presionar la tecla *enter*.

Figura 3.1. Cuadro de entrada de datos



Ejemplo 3.3. Un programa que utiliza la función *read* para capturar el nombre del usuario y mostrar un saludo.

Cuadro 3.3. Ejemplo de la función *read*

```
1.   #|Programa: prog033
2.     Función : ejemplos de la función read
3.   ----------------------------------------|#
4.   (begin
5.     (display "\n Ingrese su nombre: ")
6.     (define nom (read))
7.     (newline)
8.     (display "Bienvenido ")
9.     (display nom)
10.   )
```

```
Ingrese su nombre: Ricardo

Bienvenido Ricardo
```

En el código del cuadro 3.3 se aprecia el funcionamiento básico de una lectura de datos por teclado. En la línea 7 se imprime un mensaje solicitando el dato, en la línea 8 se utiliza la función *read* para leer y la función *define* para asociar el dato capturado con una variable.

3.2.2 Función *read-char*

Esta función es similar a *read*, pero su uso es más especializado, pues solo permite la lectura de caracteres. Su sintaxis es:

```
(read-char [puerto])
```

(read-char)

El puerto puede omitirse puesto que es un argumento opcional, en cuyo caso la lectura se hará desde el teclado.

La función *read-char* se especializa en la lectura de caracteres, de manera que si se introduce una cadena o un número de varias cifras, esta función solo captura el primer carácter. Ahora bien, dado que solo lee caracteres no es necesario especificar el formato #\. Si se digita una cadena, un número o cualquier otra expresión, la función solo tomará el primer carácter.

Ejemplo 3.4. Un programa que utiliza la función *read-char* para capturar un caracter.

Cuadro 3.4. Ejemplo de la función *read-char*

```
1.   #|Programa: prog034
2.     Función : ejemplos de la función read-char
3.   ------------------------------------------|#
4.   (begin
5.     (display "\n Ingrese un caracter: ")
6.     (define car (read-char))
7.     (newline)
8.     (display "El caracter leido es: ")
9.     (display car)
10.    (display "\n Es carácter? ")
11.    (display (char? car))
12.   )
Ingrese un caracter: 1234
El caracter leído es: 1
Es caracter? #t
```

En el cuadro 3.4, se lee un carácter y se lo asocia a la variable *car* (línea 8) puara luego mostrar y evaluarlo con la función *char?*. Al ejecutar el programa de ejemplo se introdujo 1234, se observa que solo se guardó el primer carácter.

3.2.3 Función *read-line*

Esta función se especializa en leer cadenas de caracteres. Cualquier secuencia de caracteres, incluidos números y los símbolos que especifican tipos, se toma como cadena. Como cualquier entrada que se haga con esta función será tomada como cadena, no es necesario utilizar comillas dobles.

Su sintaxis es:

```
(read-line [puerto]
(read-line)
```

Al igual que las funciones *read* y *read-char*, el puerto es un argumento opcional, si no se especifica se leerá desde el dispositivo predeterminado: teclado.

Ejemplo 3.5. Un programa que utiliza la función *read-line* para capturar una cadena.

Cuadro 3.5. Ejemplo de la función *read-line*

```
1.     Función : ejemplo de la función read-line
2.   ------------------------------------------|#
3.   (begin
4.     (display "\n Ingrese una cadena: ")
5.     (define cad (read-line))
6.     (newline)
7.     (display " La cadena leida es: ")
8.     (display cad)
9.     (display "\n Es caracter? ")
10.    (display (char? cad))
11.    (display "\n Es número? ")
12.    (display (number? cad))
13.    (display "\n Es cadena? ")
14.    (display (string? cad))
15.   )
```

```
Ingrese una cadena: 1234
 La cadena leida es: 1234
 Es caracter? #f
 Es numero? #f
 Es cadena? #t
```

En el programa del cuadro 3.5 se lee una cadena con la función *readline* (línea 8), se la imprime y luego se evalúa para determinar si es caracter, número o cadena. Al entrar el número 1234 se almacena como cadena y al evaluarlo con la función *number?* obtiene falso (#f).

Ejemplo 3.6. Un programa que lea dos números, los sume y muestre el resultado.

En el código del cuadro 3.6 se utiliza la función de entrada *read* para capturar dos números (líneas 6 y 8), pero antes de hacer la lectura se muestra mensajes utilizando la función *display* (líneas 5 y 7). Seguidamente se realiza un cálculo con los datos leídos, se utiliza la función *newline* para imprimir un saldo de línea y se muestra la información de salida utilizando nuevamente la función *display* (líneas 11 - 15).

Cuadro 3.6. Programa para sumar dos números

```
1.  #|Programa: prog036
2.    Funcionalidad: lee y suma dos números
3.  ----------------------------------------|#
4.  (begin
5.    (display "Ingrese un numero: ")
6.    (define x (read))
7.    (display "Ingrese otro numero: ")
8.    (define y (read))
9.    (define r (+ x y))
10.   (newline)
11.   (display x)
12.   (display " + ")
13.   (display y)
14.   (display " = ")
15.   (display r)
16.   )
```

```
    Ingrese un numero: 3
    Ingrese otro numero: 4

    3 + 4 = 7
```

Como se aprecia en los ejemplos presentados en este capítulo, las funciones de entrada y salida se combinan para construir la interfaz de usuario a través de la cual se introducen datos al programa y se obtiene los resultados.

Ejemplo 3.7. Un programa para calcular el área y el perímetro de un círculo.

Antes de escribir código, indiferente del lenguaje que se utilice, es necesario tener claridad sobre lo que el programa debe hacer y qué datos requiere para hacerlo. En este caso el programa debe calcular el área y el perímetro de un círculo y para ello es necesario conocer el radio, entonces éste es un dato de entrada, mientras que los dos resultados esperados son los datos de salida. Ahora que se ha identificado los datos de entrada y los de salida, en el modelo funcional, es necesario establecer la relación entre ellos: una función.

Entonces, se define *r* como el radio, *f* como el área en función del radio y *g* como el perímetro en función del radio se tiene:

$f(r) = pr^2$

$g(r) = 2pr$

Al implementar en Scheme esta solución se tiene el código que se presenta en el cuadro 3.7. (Para más información sobre la implementación de funciones ir a 2.4.6)

En el código del cuadro 3.7 se muestra la definición de funciones, que es la esencia de la programación funcional y con Scheme (líneas 6 y 10) y la utilización de las funciones de entrada y salida de datos (líneas 15 - 20).

Ejemplo 3.8. Un programa para calcular el valor futuro de una inversión de *x* capital a una tasa de interés *i* durante *n* periodos de tiempo. La solución se presenta en el cudro 3.8.

Cuadro 3.7. Programa para calcular área y perímetro de un círculo

```
1.   #|Programa: prog037
2.     Función: calcula el área y el perímetro de un
     circulo
3.   ------------------------------------------------|#
4.   (define pi 3.1416)     ;definición de variable global
5.   ;-----------------------------------------------------
6.   (define f              ;función para obtener el área
7.     (λ (r)
8.       (* pi r r)))
9.   ;-----------------------------------------------------
10.  (define g           ;función para obtener el perímetro
11.    (λ (r)
12.      (* 2 pi r)))
13.  ;-----------------------------------------------------
14.  (begin                ; bloque principal del programa
15.    (display "Ingrese el radio: ")
16.    (define r (read))
17.    (display "\n Área = ")
18.    (display (f r))
19.    (display "\n Perímetro = ")
20.    (display (g r))
21.    )
```

```
Ingrese el radio: 4
 Área = 50.2656
 Perímetro = 25.1328
```

Para solucionar este problema hay que tener en cuenta que el valor futuro de un capital se calcula en función de tres variables: el valor actual de la inversión (x), la tasa de interés (i) y el tiempo (n) que representa periodos que pueden ser: meses, trimestres, semestre o años.

Lo anterior indica que el programa en cuestión requiere que se le suministren tres datos y como resultado de las operaciones devolverá uno, el valor futuro. Habiendo identificado las entradas y las salidas, es preciso establecer las funciones.

Sea f el interés en función del valor actual, la tasa de interés y el tiempo, y g el valor futuro en función del valor actual y el interés, entonces:

*f(x,i,n) = x *(i/100) * n*
g(x,i, nf) = x + f(x, i, n)

En el código del cuadro 3.8 se utiliza las funciones de entrada y salida de datos en el bloque principal del programa (líneas 14 - 21), mientras que las operaciones se realizan en las funciones definidas en las líneas 4 y 9.

Cuadro 3.8. Programa para calcular el valor futuro de una inversión

```
1.  (define interes       ; función para calcular el interés
2.    (λ (x i n)
3.      (* x n (/ i 100))))
4.  ;-----------------------------------------------------
5.  ; función para obtener el valor futuro
6.  (define valor-futuro
7.    (λ (x i n )
8.      (+ x (interes x i n))))
9.  ;-----------------------------------------------------
10. (begin                    ; bloque principal del programa
11.   (display "\n Valor actual: ")
12.   (define x (read))
13.   (display "\n Tasa de interes: ")
14.   (define i (read))
15.   (display "\n Tiempo: ")
16.   (define n (read))
17.   (display "\n Valor futuro = ")
18.   (display (valor-futuro x i n))
19. )
```

```
Valor actual: 2000000
Tasa de interés: 3
Tiempo: 6

Valor futuro = 2360000
```

Para mayor legibilidad del código se asigna nombres más nemotécnicos a las funciones. A *f(x,i,n)* se la denomina *interes* y a *g(x, f)*, *valor-futuro*.En adelante a las funciones se les dará un nombre de acuerdo a la tarea que realizan.

CAPÍTULO 4

DECISIONES

"Nadie sabe de lo que es capaz
hasta que trata de hacerlo".
Siro Publio

En la programación, como en la vida real, es necesario evaluar las circunstancias y actuar en consecuencia, pues no siempre se puede establecer por anticipado el curso de las acciones. En la ejecución de un programa, muchas expresiones estarán supeditadas al cumplimiento de determinadas condiciones; por ello, el programador ha de identificar estas situaciones y especificar, en el programa, qué hacer en cada caso. En este capítulo se presenta las funciones del lenguaje Scheme para la implementación de decisiones.

4.1 LAS DECISIONES EN PROGRAMACIÓN

Todo problema, tanto en programación como en otros ámbitos, incluye un conjunto de variables que pueden tomar diferentes valores. La presencia o ausencia de determinadas variables, así como su comportamiento, requieren acciones diferentes en la solución. Esto hace que los programas no sean secuencias simples de instrucciones, sino estructuras complejas que implementan saltos y bifurcaciones según el contenido de las variables y las condiciones que sobre ellas se han impuesto.

Por ejemplo, para realizar una división se requieren dos variables: dividendo y divisor, que pueden contener cualquier número entero o real. No obstante, la división sobre cero no está determinada y por tanto se tiene como condición que el divisor ha de ser diferente de 0. Al implementar esta operación en un programa se debe evaluar el denominador y decidir si realizar el cálculo o mostrar un mensaje de error. Si no se implementa esta decisión y al ejecutarse el programa la variable toma el valor cero se presentará un error de cálculo y la ejecución se interrumpirá.

Implementar una decisión implica evaluar una o más variables para determinar si cumple una o más condiciones y establecer un flujo de acciones para cada resultado posible (Chaves, 2004). Las acciones pueden consistir en ejecutar una o más expresiones o en omitirlas.

Una condición es una expresión relacional o lógica cuyo resultado es un dato de tipo lógico: verdadero o falso.

En programación funcional las decisiones juegan un papel muy importante ya que además de permitir programar saltos y bifurcaciones son cruciales en la implementación de funciones recursivas, como se verá en el apartado 6.2.

4.2 TIPOS DE DECISIONES

Considerando su complejidad, las decisiones se clasifican en tres grupos: simples, dobles y múltiples.

4.2.1 Decisión simple

Se denomina simple por cuanto lo que hay que decidir es si se ejecuta u omite una expresión o un conjunto de ellas, dependiendo del resultado de una condición.

```
Si condición
    expresión(es)
Fin si
```

Se evalúa la condición, si el resultado es verdadero se ejecuta la expresión, si es falso la ejecución continúa después de *fin si*.

En Scheme se tiene las funciones *when()*, *unless()* e *if()* para implementar este tipo de decisiones. Estas funciones se explican a partir de la sección 4.3.1.

Ejemplo 4.1. Función para obtener el valor absoluto de un número.

Se sabe que el valor absoluto corresponde al mismo número si este es positivo y a su inverso si es negativo, en cuyo caso se multiplica por -1. De manera que al implementar esta función hay que decidir si se multiplica el número por -1 o si se lo deja como está, esta es una decisión simple. En el cuadro 4.1 se muestra el código de esta función, la decisión se toma en la línea 4 haciendo uso de la función *when()*.

Cuadro 4.1. Función valor-absoluto

```
1.  #|Función para obtener el valor absoluto de un numero |#
2.
3.  (define valor-absoluto (λ (n)
4.                           (when (< n 0)
5.                             (set! n (* n -1)))
6.                           n))
```

```
> (valor-absoluto 4)
4
> (valor-absoluto -4)
4
>
```

4.2.2 Decisión doble

Cuando en un punto determinado del programa se tienen dos posibles rutas de ejecución supeditadas al cumplimiento de una condición se dice que se tiene una decisión doble. Estas decisiones tienen la forma.

si (condición)
    expresión-uno
si no
    expresión-dos

El programa evalúa la condición, si ésta se cumple se ejecuta expresión-uno, en caso contrario se ejecuta expresión-dos (Joyanes, 1988). Las dos expresiones vinculadas a la decisión son mutuamente excluyentes. Para este tipo de decisiones, en Scheme se utiliza la función *if()* cuya sintaxis se explica más adelante.

Ejemplo 4.2. Función para determinar si un número entero es par o impar.

Al dividir un número entero cualquiera sobre dos se presentan dos posibilidades: que la división sea exacta o que sea inexacta. En el primer caso se trata de un número par, en el segundo de uno impar, alternativas mutuamente excluyentes. El código fuente de esta función se presenta en el cuadro 4.2.

Cuadro 4.2. Función par-impar

```
1.  #|función para determinar si un número es par o impar|#
2.  (define par-impar (λ (n)
3.                       (if(=(remainder n 2) 0)
4.                          "Número par"
5.                          "Número impar")))
```

```
> (par-impar 3)
"Número impar"

> (par-impar 8)
"Número par"
>
```

4.2.3 Decisión múltiple

Se presenta cuando, en alguna parte del programa se dispone de un conjunto de tres o más alternativas de ejecución, cada una de ellas asociada al valor de una variable. Para este tipo de decisiones se utilizan variables con un conjunto discreto de posibles valores, de manera que cada uno de ellos se relaciona con una expresión o

conjunto de éstas. En Scheme se cuenta con la función *case()* para implementar estas decisiones.

Ejemplo 4.3. Función para convertir un dígito arábigo a su equivalente romano.

Los dígitos comprenden valores entre 1 y 9 inclusive y para cada uno de ellos en el sistema arábigo se tiene su equivalente en el sistema romano. Para implementar esta función se puede utilizar varias decisiones simples o dobles, pero es más eficiente utilizar una decisión múltiple, de manera que se establezca qué valor debe devolver la función según el valor del parámetro. El código del programa se muestra en el cuadro 4.3.

Cuadro 4.3. Función arábigo-romano

```
1.   #|función para convertir un número arábigo a romano |#
2.   (define arabigo-romano (λ (n)
3.                         (case n
4.                           ((1) "I")
5.                           ((2) "II")
6.                           ((3) "III")
7.                           ((4) "IV")
8.                           ((5) "V")
9.                           ((6) "VI")
10.                          ((7) "VII")
11.                          ((8) "VIII")
12.                          ((9) "IX"))))
> (arábigo-romano 3)
"III"

> (arábigo-romano 6)
"VI"
>
```

4.3 FUNCIONES PARA IMPLEMENTAR DECISIONES

4.3.1 Función *when*

Recibe como argumento una expresión relacional o lógica y si ésta reporta un valor #t (verdadero), es decir, si la condición se cumple, ejecuta las expresiones asociadas a *when*. En caso de #f simplemente se ignoran y la ejecución continúa en la siguiente función.

La sintaxis de *when* es:

```
(when (condición)
      (expresión-1)
      (expresión-2)
      . . .
      (expresión-n))
```

Ejemplo 4.4. En la tienda La Esquina se hace un descuento del 5% en compras de 10 o más unidades de un mismo producto. Para calcular el valor total se tiene la función *calcular-total*, cuyo código se presenta en el cuadro 4.4.

Cuadro 4.4 Función calcular-total

```
1. ;Función para calcular el valor total y hacer descuento
2. (define calcular-total
3.    (λ (cantidad vlr-unidad)
4.      (define vlr-total (* cantidad vlr-unidad))
5.      (when (>= cantidad 10)
6.        (set! Vlr-total (- vlr-total (* vlr-total 0.05))))
7.      Vlr-total
8.      ))
```

```
> (calcular-total 4 1000)
4000

> (calcular-total 10 1000)
9500.0
>
```

En el código del cuadro 4.4 se ha incluido la función *when* para decidir si se aplica el descuento del 5%. La expresión de la línea 6 se ejecuta únicamente cuando la cantidad facturada es igual o superior a 10.

Ejemplo 4.5. En la empresa MIR se necesita un programa que permita calcular el valor de una bonificación para sus empleados. Esta es equivalente al 50% del sueldo mensual, pero para aquellos que hayan trabajado más de 20 años ésta se incrementa en un 30%. La función para calcular la bonificación se presenta en el cuadro 4.5.

Cuadro 4.5. Función para calcular bonificación

```
1. (define calcular-bonif
2.   (λ (sueldo antiguedad)
3.     (define bonificacion (/ (* sueldo 50) 100))
4.     (when (> antiguedad 20)
5.       (set! bonificacion (+ bonificacion (/ (*
                                  bonificacion 30) 100))))
6.     Bonificacion
7.     )
8.   )
```

```
> (calcular-bonif 500000 10)
250000
> (calcular-bonif 1000000 25)
650000
```

Ejemplo 4.6. En la EPS Zdarobia se necesita un programa que calcule la cuota moderadora para sus usuarios. Esta tiene un valor de $ 10.000 para todos los usuarios, excepto para aquellos cuyos ingresos son mayores a 4 salarios mínimos, a quienes se les incrementa en un 1% sobre el valor de su sueldo. El cuadro 4.6 presenta la solución.

4.3.2 Función *unless*

Esta función recibe como argumento una condición y la evalúa. Si el resultado es #f (falso) ejecuta las expresiones asociadas, pero si es #t (verdadero) se omiten y la ejecución continúa después de cerrar *unless*. Su comportamiento es similar a *when*, pero con una actitud

diferente frente a la condición, mientras que *when* ejecuta las expresiones cuando la condición se cumple, *unless* las omite.

Cuadro 4.6. Programa para calcular cuota moderadora

```
1.   (begin
2.     (display "\nValor del salario Minimo: ")
3.     (define sml (read))
4.     (display "\nSueldo del usuario: ")
5.     (define suel (read))
6.     (define cuota 10000.0)
7.     (when (> suel (* 4 sml))
8.       (set! Cuota (+ cuota (/ suel 100))))
9.     (display "\nValor cuota moderadora: ")
10.    (display cuota))
```

```
Valor del salario Minimo: 461500
Sueldo del usuario: 1500000
Valor cuota moderadora: 10000.0
```

La sintaxis de *unless* es:

```
(unless (condición)
      (expresión-1)
      (expresión-2)
      . . .
      (expresión-n))
```

Ejemplo 4.7. Retomando el problema del ejemplo 4.4 se tiene: en la tienda La Esquina se hace un descuento del 5% en todas las compras, a menos que la cantidad comprada sea menor a 10 unidades. Para calcular el valor total se tiene la función *calcular-total-2*, cuyo código se presenta en el cuadro 4.7.

Ejemplo 4.8. Función para determinar si un número *m* es múltiplo de *n*. La solución a este problema se presenta en el cuadro 4.8.

Cuadro 4.7. Función calcular-total-2

```
1.  #|Función para calcular el valor total y hacer descuento
    utilizando unless |#
2.  (define calcular-total-2
3.    (λ (cantidad vlr-unidad)
4.      (define vlr-total (* cantidad vlr-unidad))
5.      (unless (< cantidad 10)
6.        (set! vlr-total (- vlr-total (* vlr-total 0.05))))
7.      vlr-total
8.      ))
9.
```

```
> (calcular-total-2 12 1000)
11400.0

> (calcular-total-2 8 1000)
8000
>
```

Un número *m* es múltiplo de *n* si existe un número entero *x* que al multiplicarlo por *n* dé como resultado *m*, de la forma: *x*n = m*.

Cuadro 4.8. Función múltiplo

```
1.  ; Función para determinar si un número es múltiplo de
    otro
2.  (define multiplo
3.    (λ (m n)
4.      (unless (> (remainder m n) 0)
5.        "Es multiplo"
6.        )
7.      )
8.    )
```

```
> (multiplo 10 2)
"Es multiplo"
>
> (multiplo 45 3)
"Es multiplo"
```

La estrategia para solucionar este problema consiste en dividir de forma entera *m/n*. Si la división es exacta, es decir, si el resultado es 0, *m* es múltiplo de *n*.

En este ejemplo se utiliza la función *unless* para evaluar si hay residuo, si lo hay el primer número no es múltiplo del segundo, pero no se muestra ningún mensaje ya que esta función es para decisiones simples.

Ejemplo 4.9 Función para validar que el divisor sea diferente de cero antes de realizar una división.

Si al hacer una división el divisor es 0 se produce un error y el programa se cancela, para evitar esta situación es conveniente asegurarse de que el divisor sea diferente de 0 antes de efectuar la división. La solución se presenta en el cuadro 4.9.

Cuadro 4.9. Función evalúa-divisor

```
1.  ; Función para validar el divisor
2.  (define evalua-divisor
3.    (λ (x y)
4.      (unless (= y  0)
5.        (/ x y)
6.        )
7.      )
8.  )
```

```
> (evalua-divisor 25 5)
5
> (evalua-divisor 25 0)
>
```

En la función del cuadro 4.9 la división se efectúa a menos que el divisor (*y*) sea 0.

4.3.3 Función *if*

Es la función selectiva por excelencia, recibe una expresión condicional como argumento, la evalúa y dependiendo de su resultado ejecuta una de las expresiones especificadas. Permite que se especifique una o dos expresiones, según se trate de una decisión simple o doble.

Como decisión simple tiene la forma:

```
(if (condición)
      (expresión))
```

Si el resultado de evaluar la condición es #t se ejecuta la expresión, en caso contrario se omite y la ejecución del programa continúa después de cerrar la función *if*.

También puede utilizarse para implementar decisiones dobles, de la forma:

```
(if (condición)
      (expresión-si-verdadero)
      (expresión-si-falso))
```

Si al evaluar la condición se obtiene #t (verdadero) se ejecuta *expresión-si-verdadero*, si se obtiene #f (falso) se ejecuta *expresión-si-falso*.

Esta función solo admite que se especifique una expresión para verdadero y una para falso. Si se requiere ejecutar más de una expresión es necesario agruparlas en un bloque con la función *begin*, de la forma:

```
(if (condición)
      (begin       ; si verdadero
            (expresión-1.1)
            (expresión-1.1)
            (expresión-1.1))
      (begin       ; si falso
            (expresión-2.1)
            (expresión-2.1)
            (expresión-2.1)))
```

Ejemplo 4.10. Función para determinar el auxilio de transporte de un empleado.

Esta función recibe como parámetros el sueldo del empleado y el valor del salario mínimo. Si el sueldo es menor o igual a dos salarios

mínimos el empleado recibe el auxilio, que para este ejemplo se estima en $60.000. El cuadro 4.10 presenta el código.

Cuadro 4.10. Función auxilio-transporte

```
1.  #|Función para determinar el valor del auxilio de
    transporte |#
2.  (define auxilio-transporte
3.    (λ (sb sml)
4.      (define aux 0)
5.      (if (<= sb (* 2 sml))
6.          (set! aux 60000))
7.      Aux))
```

```
> (auxilio-transporte 700000 460000)
60000

> (auxilio-transporte 1000000 460000)
0
```

Ejemplo 4.11 Función para identificar el mayor de dos números. Considerando que los números sean diferentes entre sí, se evalúa si el primero es mayor que el segundo, si no se asume que el segundo es mayor que el primero. El cuadro 4.11 presenta el código fuente.

Cuadro 4.11. Función número-mayor

```
1.  #|Función para identificar el mayor de dos números |#
2.  (define numero-mayor
3.    (λ (a b)
4.      (define mayor 0)
5.      (if (> a b)
6.          (set! mayor a)
7.          (set! mayor b))
8.      mayor))
```

```
> (numero-mayor 5 8)
8

> (numero-mayor 15 8)
15
```

Ejemplo 4.12 Función para realizar una división. Considerando que la división sobre 0 no está determinada se trata de evitar que el

programa aborte al realizar el cálculo, para ello se valida el divisor, si este es 0 se devuelve un dato nulo.

El código se muestra en el cuadro 4.12.

Cuadro 4.12. Función división

```
1. #|Función para realizar una división |#
2. (define dividir
3.   (λ (dividendo divisor)
4.     (if (= divisor 0 )
5.         Null
6.         (/ dividendo divisor))))
```

```
> (dividir 10 5)
2

> (dividir 10 0)
()
```

Ejemplo 4.13. Un programa para calcular el área de un triángulo o de un círculo, según preferencia del usuario.

Cuadro 4.13. Programa para calcular áreas de triángulo y círculo

```
1.   #|Programa para calcular el área de un círculo o un
     triángulo|#
2.
3.   (define area-triangulo
4.     (λ (base altura)
5.       (/ (* base altura) 2)))
6.
7.   (define area-circulo
8.     (λ (radio)
9.       (* 3.1416 radio radio)))
10.
11.  (begin
12.    (display "Seleccione figura: [1] Triangulo [2]
     Círculo ")
13.    (define figura (read))
14.    (define base 0)
15.    (define altura 0)
16.    (define radio 0)
17.    (if (= figura 1)
18.        (begin
```

Cuadro 4.13. (Continuación)

```
19.          (display "\nTriángulo")
20.          (display "\nBase: ")
21.          (set! base (read))
22.          (display "\nAltura: ")
23.          (set! altura (read))
24.          (display "\nArea: ")
25.          (display (area-triangulo base altura)))
26.        (begin
27.          (display "\nCírculo")
28.          (display "\nRadio: ")
29.          (set! radio (read))
30.          (display "\nArea: ")
31.          (display (area-circulo radio)))))
```

```
Seleccione figura: [1] Triangulo [2] Círculo  1

Triángulo
Base: 4
Altura: 6
Área: 12

Seleccione figura: [1] Triangulo [2] Círculo  2

Círculo
Radio: 2
Área: 12.5664
```

En el ejemplo de cuadro 4.13 se utiliza una función *if* en la que tanto para el resultado verdadero, como para el falso, se tienen varias expresiones agrupadas mediante la función *begin* (líneas 18 y 26).

4.3.4 Función *case*

Esta función evalúa una variable y dependiendo de su valor ejecuta una expresión o un conjunto de ellas, de esta manera permite la implementación de decisiones múltiples. Si el contenido de la variable no coincide con alguno de los valores previstos se ejecuta la expresión vinculada con la palabra *else*.

Su sintaxis es:

```
(case variable
        ((valor-1) (expresión-1))
        ((valor-2) (expresión-2)
        ((valor-3) (expresión-3))
        ((valor-4) (expresión-4)
        (else (expresión-n)))
```

Si el contenido de *variable* es *valor-1* se ejecuta *expresión-1* y termina la función *case*, si *variable* contiene *valor-2* se ejecuta *expresión-2* y termina, y así sucesivamente. Todas las opciones son mutuamente excluyentes.

Ejemplo 4.14 Función que recibe el número del día y devuelve su nombre.

Cuadro 4.14. Función nombre-día

```
1.  #|Función para conocer el nombre de un día de la
    semana|#
2.  (define nombre-dia
3.    (λ (numdia)
4.      (case numdia
5.        ((1) "Lunes")
6.        ((2) "Martes")
7.        ((3) "Miércoles")
8.        ((4) "Jueves")
9.        ((5) "Viernes")
10.       ((6) "Sábado")
11.       ((7) "Domingo")
12.       (else "Número no válido"))))
```

```
> (nombre-dia 1)
"Lunes"

> (nombre-dia 3)
"Miércoles"

> (nombre-dia 9)
"Número no válido"
```

Ejemplo 4.15 Una función para determinar el número de días que tiene un mes.

Los meses tienen 30 y 31 días, excepto febrero que puede tener 28 o 29 dependiendo si se trata de un año bisiesto o no. Para que el resultado de la función sea exacto es necesario que se le proporcionen dos argumentos, el número del mes y el año. El código se presenta en el cuadro 4.15.

Cuadro 4.15. Función número-días

```
1.   ; funcion para determinar el numero de dias de un año
2.   (define numero-dias
3.     (lambda (anio mes)
4.       (case mes
5.         ((1) 31)
6.         ((2) (if(and(=(remainder anio 4) 0)
                       (or(not(=(remainder anio 100) 0))
                          (=(remainder anio 400) 0)))
7.                  29
8.                  28))
9.         ((3) 31)
10.        ((4) 30)
11.        ((5) 31)
12.        ((6) 30)
13.        ((7) 31)
14.        ((8) 31)
15.        ((9) 30)
16.        ((10) 31)
17.        ((11) 30)
18.        ((12) 31)
19.        )
20.      )
21.    )
```

```
> (numero-dias 2004 2)
29
> (numero-dias 2100 2)
28
> (numero-dias 2009 3)
31
> (numero-dias 2009 2)
28
> (numero-dias 2009 4)
30
```

Ejemplo 4.16. Un programa que mediante un menú efectúe las operaciones aritméticas básicas sobre dos números enteros.

Un menú consiste en un listado de opciones asociadas a tareas que el programa realiza. Esta estrategia de programación es muy común, porque facilita el control de la interacción con el usuario a la vez que facilita el manejo del programa. Normalmente, un menú se implementa utilizando una estructura iterativa y una decisión múltiple; pero en este ejemplo solo se utiliza la decisión ya que las iteraciones se estudian en el capítulo siguiente. Ver código en el cuadro 4.16

Cuadro 4.16. Menú para las operaciones aritméticas básicas

```
1.  ; menu para realizar operaciones aritméticas básicas
2.  (begin
3.    (display "\n Primer número:")
4.    (define x (read))
5.    (display "\n Segundo número:")
6.    (define y (read))
7.    (display "\n Operación a realizar:")
8.    (display "\n\t 1. Suma")
9.    (display "\n\t 2. Resta")
10.   (display "\n\t 3. Multiplicación")
11.   (display "\n\t 4. División")
12.   (display "\n Seleccione una opción:")
13.   (define opc (read))
14.   (case opc
15.     ((1) (display x)
16.          (display " + ")
17.          (display y)
18.          (display " = ")
19.          (display (+ x y)))
20.     ((2) (display x)
21.          (display " - ")
22.          (display y)
23.          (display " = ")
24.          (display (- x y)))
25.     ((3) (display x)
26.          (display " * ")
27.          (display y)
28.          (display " = ")
29.          (display (* x y)))
30.     ((4) (if(not(= y 0))
```

Cuadro 4.16. (Continuación)

```
31.              (begin
32.                (display x)
33.                (display " / ")
34.                (display y)
35.                (display " = ")
36.                (display (/ x y)))
37.              (display "\n No se puede dividir sobre 0")))
38.      (else (display "\n Operacion no válida")))
39.    )
```

```
Primer número:45
Segundo número:15
Operación a realizar:
        1. Suma
        2. Resta
        3. Multiplicación
        4. División
 Seleccione una opción:1
 45 + 15 = 60
```

4.4 DECISIONES ANIDADAS

Se denomina *anidadas* a las decisiones que se escriben una dentro de otra, de la forma:

```
Si condición-1
    Si condición-2
        Expresión-1
    Si no
        Expresión-2
    Fin si
Si no
    Si condición-3
        Expresión-3
    Fin si
Fin si
```

Si condición-1 es verdadera se evalúa condición-2 y si esta también es verdadera se ejecuta expresión-1. De forma que expresión-1 se ejecuta si condición-1 y condición-2 son verdaderas, expresión-2 si condición-1 es verdadera y condición-2 es falsa.

Cuando la decisión anidada se escribe para el caso de que la primera sea verdadera se dice que se anida por verdadero, como ocurre con la condición-2. Si la segunda decisión se evalúa cuando la primera no se cumple, se dice que se anida por falso. Tanto por verdadero como por falso se puede anidar todas las decisiones que sea necesario.

4.4.1 Anidamiento con *when*, *unless*, *if* y *case*

En Scheme, las funciones *when*, *unless*, *if* y case pueden anidarse indistintamente. Es decir, dentro de una función *when* se puede escribir otra *when* o cualquiera de las otras.

Ejemplo 4.17. Una función para determinar si un año es bisiesto

Se conoce que un año es bisiesto cuando siendo divisible para 4 no lo es para 100 o cuando es divisible para 400. En el cuadro 4.17 se presenta la solución.

Cuadro 4.17 Función año-bisiesto

```
1.   ; Función para determinar si un año es bisiesto
2.   (define año-bisiesto
3.     (λ (x)
4.       (define bisiesto #f)
5.       (unless(>(remainder x 4) 0)
6.          (if(=(remainder x 100) 0)
7.             (when(=(remainder x 400) 0)
8.               (set! bisiesto #t))
9.             (set! bisiesto #t)))
10.        bisiesto))
```

```
> (año-bisiesto 1999)
#f
>  (año-bisiesto 1700)
#f
>  (año-bisiesto 2008)
#t
```

Como se aprecia en las líneas 5, 6 y 7, en este ejemplo se utiliza las tres funciones de manera anidada.

Ejemplo 4.18. Una función que recibe tres números diferentes e identifica el mayor de ellos. El código de la función se muestra en el cuadro 4.18.

Cuadro 4.18 Función número-mayor

```
1. ; Función para el mayor de tres números diferentes
2. (define numero-mayor
3.    (λ (a b c)
4.       (if(> a b)
5.         (if(> a c)
6.            A
7.            c)
8.         (if(> b c)
9.            B
10            c))))
```

```
> (numero-mayor 2 3 4)
4
> (numero-mayor 10 7 4)
10
> (numero-mayor 2 8 5)
8
```

4.4.2 Función cond

La función *cond* permite el anidamiento de decisiones mediante el establecimiento de condiciones que serán evaluadas consecutivamente hasta encontrar una que sea verdadera (#t). La sintaxis de *cond* es:

```
(cond
   ((condición-1) (expresión-1))
   ((condición-2) (expresión-2))
   ((condición-3) (expresión-3))
   ((condición-4) (expresión-4))
   (else (otra-expresión)))
```

Cada expresión está asociada al cumplimiento de la condición y se ejecuta de manera exclusiva. Es decir, si *condición-1* es verdadera se ejecuta *expresión-1* y ya no se evalúan las demás condiciones.

Esta estructura de decisiones implementada mediante la función if tendría la forma:

```
(if (condición-1)
       (expresión-1)
       (if (condición-2)
              (expresión-2)
              (if (condición-3)
                     (expresión-3)
                     (if (condición-4)
                            (expresión-4)
                            (otra-expresión)))))
```

Ejemplo 4.19. La empresa Buen Patrón desea aumentar los sueldos de sus empleados de manera inversamente proporcional a la cantidad de salarios mínimos que devengan. Los incrementos se realizarán según la siguiente tabla:

Sueldo	Incremento
Menor o igual a dos sm.	10%
Mayor a dos y hasta 4 sm.	8%
Mayores a 4 y hasta 6 sm.	6%
Más de 6 sm.	4%

Se requiere una función para calcular el nuevo salario de cada empleado.

La solución de este ejercicio requiere la aplicación de decisiones anidadas. En el cuadro 4.19 se muestra el código fuente de esta función en la que se utiliza la función *cond*.

Cuadro 4.19. Función nuevo-sueldo utilizando cond

```
1.   ; función para calcular el nuevo sueldo de un empleado
2.   (define nuevo-sueldo
3.     (λ (sueldo sm)
4.       (cond
5.         ((<= sueldo (* sm 2)) (+ sueldo (* sueldo 0.1)))
6.         ((<= sueldo (* sm 4)) (+ sueldo (* sueldo 0.08)))
7.         ((<= sueldo (* sm 6)) (+ sueldo (* sueldo 0.06)))
8.         (else(+ sueldo (* sueldo 0.04))))))
```

Cuadro 4.19. (Continuación)

```
> (nuevo-sueldo 600000 461500)
660000.0
> (nuevo-sueldo 1000000 461500)
1080000.0
> (nuevo-sueldo 2000000 461500)
2120000.0
> (nuevo-sueldo 4000000 461500)
4160000.0
```

La solución a este mismo ejercicio anidando con *if* se muestra en el cuadro 4.20.

Cuadro 4.20. Función nuevo-sueldo utilizando *if*

```
1.   ; función para calcular el nuevo sueldo de un empleado
2.   (define nuevo-sueldo
3.     (λ (sueldo sm)
4.        (if(<= sueldo (* sm 2))
5.           (+ sueldo (* sueldo 0.1))
6.           (if (<= sueldo (* sm 4))
7.               (+ sueldo (* sueldo 0.08))
8.               (if (<= sueldo (* sm 6))
9.                   (+ sueldo (* sueldo 0.06))
10.                  (+ sueldo (* sueldo 0.04)))))))
```

```
> (nuevo-sueldo 600000 461500)
660000.0
> (nuevo-sueldo 1000000 461500)
1080000.0
> (nuevo-sueldo 4000000 461500)
4160000.0
```

CAPÍTULO 5

ITERACIONES

"Solo la imaginación me dice
lo que puede ser".
A. Camus

Estas estructuras también se conocen como ciclos, bucles o estructuras de repetición. Consisten en alterar la secuencia de ejecución del programa haciendo que una expresión, o un conjunto de ellas, se repita un número determinado de veces, que estará determinado por el comportamiento de una variable y la evaluación de una condición.

Por ejemplo, si se requiere procesar las notas de 50 estudiantes, esto implica operaciones como: la lectura de las notas parciales, el cálculo e impresión de la nota definitiva, para cada uno de ellos. Podría escribirse 50 instrucciones de entrada, 50 cálculos y 50 instrucciones de salida, pero esta no es una buena solución; en su lugar, se escribe las instrucciones para un estudiante y se colocan en una estructura de repetición de manera que puedan ejecutarse las mismas para todo el grupo.

El concepto de iteración es propio del modelo imperativo, el funcional aplica el concepto de recursividad. Scheme permite la construcción de funciones recursivas y también la implementación de procedimientos recursivos al interior de un programa; estos últimos a simple vista son similares a los ciclos, pero en realidad se ejecutan de forma recursiva.

Por otra parte, por ser Scheme un lenguaje híbrido incluye la forma especial *do*, que es una estructura realmente iterativa (Appleby y Vandekopple, 1998).

En este capítulo se presenta los conceptos de contador y acumulador que son importantes tener control sobre repeticiones y dos formas de implementar iteraciones en Scheme, la primera utilizando procedimientos recursivos a través de la forma especial *let* y la segunda utilizando la estructura iterativa *do*.

5.1 CONTADORES Y ACUMULADORES

Al trabajar con ciclos es común utilizar variables de tipo contador y acumulador. (Chaves, 2004) las define así:

5.1.1 Contadores

Son variables que se utilizan para registrar el número de veces que se ejecuta una operación o un grupo de ellas. El uso más frecuente es como variable de control de un ciclo finito, en cuyo caso guardará el número de iteraciones, pero también pueden contener el número de registros de un archivo o la cantidad de datos que cumplen una condición.

Los contadores se incrementan o decrementan con un valor constante, que por lo regular es de uno en uno.

Ejemplo: se desea ingresar las notas definitivas de los estudiantes de un grupo para calcular el promedio del mismo. Dado que el tamaño del grupo puede variar, se requiere una variable (contador) para registrar el número de notas ingresadas, para luego poder calcular el promedio.

5.1.2 Acumuladores

También se llaman totalizadores. Son variables que se utilizan para almacenar valores que se leen o se calculan repetidas veces. Por

ejemplo, si se quisiera calcular el promedio de notas de un grupo de estudiantes, lo primero que se hará es leer las notas y cada una se sumaría en una variable (acumulador) de modo que después de leer todas las notas se dividirá la sumatoria de las mismas sobre el número de estudiantes para obtener el promedio.

En este ejemplo se ha hecho referencia a un contador para conocer el número de estudiantes y de un acumulador para sumar las notas.

El acumulador se diferencia del contador en que éste no tiene incremento regular, sino que se incrementa de acuerdo al valor leído o al resultado de una operación.

5.2 FORMA ESPECIAL *LET*

Let permite establecer un ámbito local en el que se puede utilizar variables y procedimiento (Kelsey, Clinger y Rees, 1998). Si se tiene en cuenta que Scheme puede definir un identificador para hacer referencia indistintamente a datos o funciones, se puede implementar un procedimiento dentro del cuerpo de un programa y hacer que se ejecute cuantas veces sea necesario, creando así una forma especial de iteración.

Para implementar un ciclo con *let* se procede de la forma que se muestra en el cuadro 5.1.

Cuadro 5.1. Ciclo con la forma especial *let*

```
(let nombre-ciclo (
      (variable1 valor1)
      (variable2 valor2)
      (variable-n valor-n))
      (expresión-1)
      (expresión-2)
      (expresión-n)
      (if (condicion)
         (nombre-ciclo parámetro1 parametro2 parametro-n)))
```

De esta manera se tiene un ámbito local en el que se define cero, una o más variables locales, de la misma forma que parámetros para una

función y un cuerpo conformado por una o más expresiones, finalmente se tendrá una condición para determinar si se vuelve a ejecutar el cuerpo del *let*. Para que las expresiones se repitan se invoca el identificador del ciclo con los parámetros que correspondan.

Ejemplo 5.1: Un programa que muestre los números del 1 al 10. El código Scheme se muestra en el cuadro 5.2.

Cuadro 5.2. Ejemplo de ciclo con *let*

```
1  (begin
2    (let repetir ((x 1))
3      (display x)
4      (display "\t")
5      (if (< x 10)
6          (repetir (+ x 1)))))
```

```
1      2      3      4      5      6      7      8      9
       10
```

En el ejemplo del cuadro 5.2 se define el procedimiento *repetir* mediante la forma especial *let*, dentro de éste se declara la variable x cuyo primer valor es 1 y el cuerpo consiste en mostrar el valor de *x* y hacer un salto de tabulación (líneas 4 y 5). Luego se averigua si el número mostrado es menor a 10, en cuyo caso se vuelve a ejecutar el procedimiento *repetir* y envía como parámetro *x+1*.

Ejemplo 5.2 Un programa que lea números desde el teclado y los sume. Cuando el usuario introduzca el 0 muestra el total y termina. La solución se presenta en el cuadro 5.3.

Cuadro 5.3. Programa para entrar números y sumarlos

```
1. (begin
2.   (define num 0)
3.   (let repetir ((acumulador 0))
4.     (display "\n Entre un numero:")
5.     (set! Num (read))
6.     (if (= num 0)
7.         (display acumulador)
8.         (repetir (+ acumulador num)))))
```

Cuadro 5.3. (Continuación)

```
Entre un número:5
Entre un número:4
Entre un número:3
Entre un número:2
Entre un número:0
14
```

En el programa del cuadro 5.3 se utiliza un acumulador para almacenar la sumatoria de los números leídos. Este se inicializa en 0 y se actualiza cada vez que se llama al procedimiento *repetir* (líneas 3 y 8).

Ejemplo 5.3. Programa para mostrar los divisores de un número. En el cuadro 5.4 se presenta el código de este programa.

Cuadro 5.4. Programa para encontrar los divisores de un número

```
1.   (begin
2.     (display "Numero: ")
3.     (define num (read))
4.     (display "\nDivisores:\n")
5.     (let repetir ((x num) (contador 1))
6.       (when(=(remainder x contador) 0)
7.            (display contador)
8.            (newline))
9.       (if (< contador x)
10.           (repetir x (+ contador 1)))))
Numero: 18

Divisores:
1
2
3
6
9
18
```

En este programa el ciclo recibe dos argumentos: el número para el cual se buscan los divisores y un contador que toma como valor inicial el 1 y luego se incremente de uno en uno hasta ser igual al número leído. En cada iteración se verifica si el número puede dividirse

exactamente sobre el contador en cuyo caso el valor actual del contador es un divisor del número.

Ejemplo 5.4: Un programa que lea *n* números y muestre el número mayor, el número menor y el promedio. Cuando se introduce el 0 no se leen más números.

Para solucionar este ejercicio es necesario utilizar un contador, un acumulador y dos variables más para mantener los números menor y mayor, en cada iteración. La solución se muestra en el cuadro 5.5

En este programa se definen cinco variables: *con*, *acum.*, *may*, *men y num*, al comienzo del programa y en cada iteración las actualiza utilizando la función *set!*.Si se retoma lo que se explicó en el primer capítulo se puede decir que este programa tiene un enfoque imperativo, ya que en el modelo funcional no se aplica la asignación de valores a variables.

Cuadro 5.5. Programa para encontrar el promedio y los números mayor y menor de entre *n* entradas.

```
1.   (begin
2.     (define cont 0)
3.     (define acum 0)
4.     (define may null)
5.     (define men null)
6.     (define num null)
7.     (let repetir ()
8.       (display "\nNumero:")
9.       (set! Num (read))
10.      (if(not(= num 0))
11.         (begin
12.           (if(= cont 0)
13.              (begin
14.                (set! may num)
15.                (set! men num))
16.              (cond
17.                ((> num may) (set! May num))
18.                ((< num men) (set! Men num))))
19.           (set! cont (+ cont 1))
20.           (set! acum (+ acum num))
21.           (repetir))))
```

Cuadro 5.5 (Continuación)

```
22.
23.     (cond
24.       ((= cont 0)
25.        (display "\n No se digitaron numeros..."))
26.       ((> cont 0)
27.        (display "\nNumeros entrados: ")(display cont)
28.        (display "\nPromedio: ")(display (/ acum cont))
29.        (display "\nNumero menor: ")(display men)
30.        (display "\nNumero mayor: ")(display may))))
```

```
Numero:3
Numero:4
Numero:5
Numero:6
Numero:7
Numero:0

Números entrados: 5
Promedio: 5
Numero menor: 3
Numero mayor: 7
```

En el cuadro 5.6 se muestra la solución al mismo problema aplicando el modelo funcional.

Cuadro 5.6. Programa para encontrar el promedio y los números mayor y menor de entre *n* entradas.

```
1.    (define mayor (λ (x y)
2.                    (if(> x y) x y)))
3.    (define menor (λ (x y)
4.                    (if(< x y) x y)))
5.    (begin
6.      (display "\nNumero:")
7.      (let  repetir  ((cont  0)(acum  0)(may  0)(men  0)(num
                        (read)))
8.        (if(= num 0)
9.           (cond
10.            ((= cont 0)
11.             (display "\n No se digitaron numeros..."))
12.            ((> cont 0)
13.             (display  "\nNumeros  entrados:  ")(display
                cont)
14.             (display  "\nPromedio:  ")(display  (/  acum
                cont))
```

Cuadro 5.6. (Continuación)

```
15.            (display "\nNumero menor: ")(display men)
16.            (display "\nNumero mayor: ")(display may)))
17.         (if(= cont 0)
18.             (repetir (+ cont 1)(+ acum num) num num
                (read))
19.            (repetir (+ cont 1)(+ acum num) (mayor num
                              may)(menor num men)(read))))))
```

En este programa se utiliza dos funciones para identificar los números mayor y menor y se colocan como argumentos en la invocación del ciclo. Las variables se definen como locales a *let* en cada iteración. Si se compara el código de los cuadros 5.5 y 5.6 se aprecia que es significativamente más corto y más elegante el que aplica el modelo funcional que el del modelo imperativo.

Ejemplo 5.5. Un programa para calcular el promedio de cuatro notas validando que cada nota ingresada esté entre 0 y 5.0. Ver la solución en el cuadro 5.7

Cuadro 5.7. Programa para validar notas y calcular el promedio.

```
1.   (begin
2.     (define suma 0)
3.     (define nota 0)
4.     (let repetir ((cont 1))
5.       (display "\n Nota ")
6.       (display cont)
7.       (display ": ")
8.       (set! nota (read))
9.       (if(or(> nota 5)(< nota 0))
10.         (begin
11.           (display " Nota no valida")
12.           (repetir cont))
13.         (begin
14.           (set! suma (+ suma nota))
15.           (when(< cont 4)
16.             (repetir (+ cont 1))))))
17.    (display "\n Promedio: ")
18.    (display (/ suma 4)))
```

Cuadro 5.7. (Continuación)

```
Nota 1: 3.5

Nota 2: 4.3

Nota 3: 5.5
Nota no valida
Nota 3: 5

Nota 4: -2.0
Nota no valida
Nota 4: 2.0

Promedio: 3.7
```

En el programa del cuadro 5.7 se define un procedimiento iterativo para leer cuatro notas válidas. Cuando la nota introducida está en el intervalo 0-5 se invoca el procedimiento para leer la siguiente nota, pero cuando no es válida se lee la misma.

5.3 EXPRESIÓN *DO*

Esta es una forma especial iterativa que no pertenece al modelo funcional (Sitaram, 2004). Scheme, por ser un lenguaje funcional impuro, la ofrece para facilitar el trabajo de los programadores que están acostumbrados al paradigma imperativo.

Para implementar un ciclo con *do* se especifica una variable, su valor inicial, el cambio en cada iteración y la condición de salida. En cada iteración puede ejecutar una o más expresiones. Conviene insistir en que la condición que se especifica es de salida, es decir que las expresiones se repiten siempre que la evaluación de la condición sea #f.

El uso más sencillo de ***do*** es con una sola variable y su sintaxis es la siguiente:

```
(do   ((vble   vlr-inicial   [(expr-cambio)]))   ((cond-
salida))
      (expresiones-a-repetir)
)
```

Es obligatorio que se especifique al menos una variable, su valor inicial y la condición de salida, la expresión de cambio es opcional, por ello se ha escrito entre [ ].

Ejemplo 5.6. Un programa para mostrar los números pares entre 2 y 20. La solución se presenta en el cuadro 5.8

Cuadro 5.8. Programa para mostrar los números pares entre 2 y 20

```
1. (begin
2.   (do((x 2 (+ x 2)))((> x 20))
3.     (display x)
4.     (display "\t")))
```

```
2     4     6     8     10    12    14    16    18
      20
```

Al utilizar la función *do* se definen automáticamente una o más variables locales, en este caso se define la variable *x* y se le asigna el valor inicial *2*, se especifica que en cada iteración se sumará 2 al valor de *x* y que la iteración terminará cuando *x > 20*. En cada iteración se mostrará el valor de x; por tanto, se mostrará los números del 2 al 20.

Ejemplo 5.7. Un programa para mostrar la tabla de multiplicar de un número cualquiera. La solución se muestra en el cuadro 5.9.

En una expresión *do* se puede definir más de una variable, ya sea que se especifique una expresión de cambio o no y, de igual manera, la condición de salida puede incluir una o más de las variables definidas. En el ejemplo cuadro 5.9 se definen e inicializan dos variables, pero solo una se incrementa y solo una establece la condición de salida.

Cuadro 5.9. Programa para mostrar la tabla de multiplicar de un número *n*.

```
1.   (begin
2.     (display "Tabla del: ")
3.     (do ((m (read))(n 1 (+ n 1))) ((> n 10))
4.         (newline)
5.         (display m)
6.         (display " * ")
7.         (display n)
8.         (display " = ")
9.         (display (* m n))
10.      )
11.    )
```

```
Tabla del: 7

7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
```

Ejemplo 5.8. Un programa para calcular el promedio de notas de un grupo de *n* estudiantes.

En este ejemplo se utiliza un contador y un acumulador. El primero se define dentro del ciclo y solo existe en ese ámbito, mientras que el segundo es utilizado dentro y fuera del bucle. La solución se presenta en el cuadro 5.10

Ejemplo 5.9. Una función para encontrar el Mínimo Común Múltiplo (MCM) de dos números.

Para encontrar el MCM se toma el número más grande y se comienza a generar sus múltiplos de menor a mayor, cada vez que se genera uno se verifica si también es múltiplo del otro número. Cuando se

encuentra un número que es múltiplo de los dos terminan las iteraciones. El código Scheme para esta función se presenta en el cuadro 5.11.

Cuadro 5.10. Programa para calcular promedio de notas

```
1.  (begin
2.    (define nota 0)
3.    (define acu 0)
4.    (display "Cuantos estudiantes?: ")
5.    (define num-est (read))
6.      (do ((con 1 (+ con 1))) ((> con num-est))
7.          (display "\n Ingrese nota del estudiante ")
8.          (display con)
9.          (display ": ")
10.         (set! nota (read))
11.         (set! acu (+ acu nota)))
12.     (display "\n Promedio del grupo: ")
13.     (display (/ acu num-est)))
```

```
Cuantos estudaintes?: 4
 Ingrese nota del estudiante 1: 3.5
 Ingrese nota del estudiante 2: 4.0
 Ingrese nota del estudiante 3: 4.2
 Ingrese nota del estudiante 4: 4.5
 Promedio del grupo: 4.05
```

Cuadro 5.11. Función para encontrar el MCM de dos números

```
1.   (define mayor (λ (x y)
2.       (if(> x y) x y)))
3.
4.   (define mcm (λ (x y)
5.      (define mcm 0)
6.      (do ((i 1 (+ i 1)) (m (mayor x y)) (encontrado #f))
                                          ((eqv? encontrado #t))
7.           (set! mcm  (* m i))
8.           (when(and (= (remainder mcm x) 0) (= (remainder
                       mcm y) 0))
9.                (set! encontrado #t)))
10.          mcm)
11.  )
```

```
> (mcm 2 10)  10
> (mcm 12 10)  60
> (mcm 7 8)  56
```

Ejemplo 5.10. Un programa para determinar si un número es primo.

Se sabe que un número primo solo es divisible para 1 y para sí mismo, por ello para saber si el número ingresado es primo se busca un divisor mayor a 2 y menor a la raíz cuadrada del número. El código Scheme de este programa se muestra en el cuadro 5.12.

Cuadro 5.12. Programa para determinar si un número es primo.

```
1.   (begin
2.     (display "Numero: ")
3.     (define  num (read))
4.     (define primo #t)
5.     (do((i 2 (+ i 1)))((or(> i (sqrt num))(boolean=? primo
     #f)))
6.        (display "\ni: ")
7.        (display i)
8.        (if(=(remainder num i) 0)
9.           (set! primo #f)))
10.    (if(boolean=? primo #t)
11.       (display "\nEs numero primo")
12.       (display "\nNo numero primo")))
```

```
Numero: 103
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
i: 10
Es número primo

Número: 1001
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
No es número primo
```

En la línea 5 se declara el ciclo estableciendo como condición de salida que la variable i que comienza en 2 supere el valor de la raíz cuadrada

del número en cuestión o que se haya encontrado un divisor mayor a 1.

Estudiando cuidadosamente los 10 ejemplos que se presentan en este capítulo será fácil utilizar tanto *let* como *do* para solucionar problemas que requieren iteraciones.

CAPÍTULO 6

RECURSIVIDAD

"Nueve décimas de nuestra sabiduría consisten en ser sabios a tiempo".
Roosevelt.

Recursividad o recursión, como también se le llama, es un concepto que se aplica de manera especial en las matemáticas y en la programación de computadores; y de igual manera, en numerosas situaciones de la vida cotidiana. En matemáticas se habla de *definición inductiva* para referirse a los numerosos métodos que hacen uso de la recursividad; mientras que en programación, este tema, se aplica a los algoritmos, en el momento de la especificación, y a las funciones, en la implementación (Joyanes, 1999).

La utilización del concepto de recursividad en los lenguajes de programación se la debe al profesor John McCarthy, del Massachusetts Institute of Tenology (MIT), quien recomendó su inclusión en el diseño de Algol60 y desarrolló el lenguaje Lisp, que introdujo estructuras de datos recursivas junto con procedimientos y funciones recursivas(Baase, 2002).

La recursividad es una alternativa a la utilización de estructuras iterativas. Un algoritmo recursivo ejecuta la repetición de cálculos mediante llamados consecutivos a sí mismo. Esto no significa que los algoritmos recursivos sean más eficientes que los iterativos, incluso, en algunos casos, los programas pueden requerir más tiempo y memoria para ejecutarse, pero este costo puede ser compensado por

una solución más intuitiva y sustentada matemáticamente en una prueba por inducción.

La mayoría de los lenguajes de programación actuales permiten la implementación de algoritmos recursivos, mediante procedimientos o funciones. Para el caso de lenguajes que no permitieran su implementación directa, ésta puede simularse utilizando estructuras de datos de tipo pila.

La recursividad es un concepto que se aplica indistintamente a algoritmos, procedimientos y programas. No obstante, en este documento se hará referencia únicamente al concepto de *función* teniendo en cuenta que su implementación se presenta bajo el modelo de programación funcional.

6.1 LA RECURSIVIDAD COMO TÉCNICA DE PROGRAMACIÓN

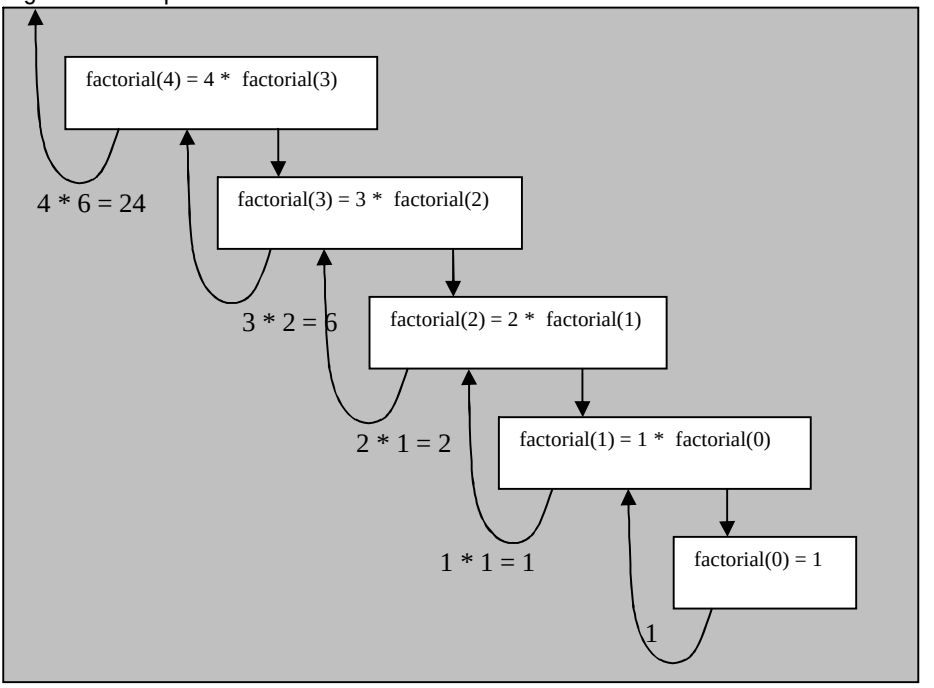
En el ámbito de la programación se define como recursividad la técnica de plantear la solución de un problema invocando consecutivamente la misma función con un problema cada vez menos complejo, hasta llegar a un problema con una solución conocida.

Por ejemplo, para calcular el factorial de 4 (4!) se le pedirá a la función que multiplique 4 por el factorial de 3 (3!); para calcular el factorial de 3 (3!), que multiplique 3 por el factorial de 2 (2!); para calcular el factorial de 2 (2!) que multiplique 2 por factorial de 1 (1!); para calcular factorial de 1 (1!) que multiplique 1 por el factorial de 0. Ahora, como se conoce que el factorial de 0 es 1 (0! = 1), entonces el problema esta resuelto.

Como se aprecia en la figura 6.1, la recursividad consiste en escribir funciones que entre sus líneas o sentencias incluyen un llamado a sí mismas con valores diferentes para sus parámetros, de manera que progresivamente llevan a la solución definitiva.

La recursividad es otra forma de hacer que una fracción de código se ejecute repetidas veces, pero no se implementa una estructura iterativa como: **mientras**, **para** o **repita hasta**.

Figura 6.1 Esquema de una función recursiva



6.2 ESTRUCTURA DE UNA FUNCIÓN RECURSIVA

Se ha mencionado que consiste en implementar funciones que se invocan a sí mismas. Esto implica que la función debe tener una estructura tal que le permita invocarse a sí misma y ejecutarse tantas veces como se requiera para solucionar el problema, pero a la vez, sin hacer más invocaciones de las estrictamente necesarias, evitando que se genere una secuencia infinita de llamadas a sí misma. Cuando una función recursiva especifica en su definición cuándo autoinvocarse y

cuándo dejar de hacerlo se dice que aquella está *correctamente definida* (Lipschutz, 1993)

Una función recursiva bien definida ha de cumplir las dos condiciones siguientes:

1. Debe existir un *criterio base* cuya solución se conoce y que no implica una llamada recursiva. En el caso de utilizar una función recursiva para calcular el factorial, el criterio base indica que factorial de 0 es 1 (0! = 1). Es decir, si la función recibe como parámetro el 0 ya no necesita invocarse nuevamente, simplemente retorna el valor correspondiente, en este caso el 1.

   factorial(n) = ? (requiere llamada a la función recursiva)
   factorial(0) = 1 (no requiere llamado a la función)

2. Cada vez que la función se invoque a sí misma, directa o indirectamente, debe hacerlo con un valor más cercano al *criterio base*. Dado que el criterio base es un caso particular del problema en el que se conoce la solución esto indica que con cada llamado a la función recursiva se estará acercando a la solución conocida. Continuando con el ejemplo de la función factorial, se ha determinado que el *criterio base* es *0! = 1*, entonces, cada vez que se invoque la función deberá utilizarse como parámetro un valor más cercano a 0.

   factorial(n) = n * factorial(n – 1)
   factorial(n – 1) = (n - 1) * factorial(n – 2)

Aplicando los criterios para una función correctamente definida se tiene que la función factorial, expresada en pseudocódigo se escribe como aparece en el cuadro 6.1. Para una explicación detallada sobre esta función consúltese la sección 6.7.2.

Cuadro 6.1 Función factorial en pseudocódigo

```
1.  función factorial(n)
2.    entero:  fac
3.    Si (n = 0) entonces
4.       Retornar  1
```

Cuadro 6.1 (Continuación)

```
5.     Si no
6.        Retornar (n * factorial(n – 1))
7.     Fin si
8.  Fin función
```

En la definición se puede apreciar claramente el cumplimiento de las dos condiciones de que trata esta sección. En las líneas 3 y 4 se examina el cumplimiento del *criterio base*, para el cual se conoce la solución (0! = 1).

```
3.     Si (n = 0) entonces
4.        Retornar 1
```

Si aun no se ha llegado a dicho criterio, es necesario invocar nuevamente la función recursiva, pero con un valor menor, como se aprecia en las líneas 5 y 6.

```
5.     Si no
6.        Retornar (n * factorial(n – 1))
```

6.3 EJECUCIÓN DE FUNCIONES RECURSIVAS

La ejecución de una función recursiva requiere que dinámicamente se le asigne la memoria suficiente para almacenar sus datos. Por cada ejecución se reserva espacio para los parámetros, las variables locales, las variables temporales y el valor devuelto por la función. El código se ejecuta desde el principio con los nuevos datos. Cabe aclarar que no se hace copia del código. (Schildt, 1993).

El espacio en que se ejecuta cada invocación de una función recursiva se denomina *Marco de activación* (Baase, 2002). Este marco, además de proporcionar espacio para guardar las variables con que opera la función, también, proporciona espacio para otras necesidades contables, como la dirección de retorno, que indica la instrucción que

se ha de ejecutar una vez salga de la función recursiva. Así, se crea un marco de referencia en el que la función se ejecuta únicamente durante una invocación.

Ahora bien, como la función será invocada un número indeterminado de veces (depende del valor de los parámetros) y por cada invocación se creará un *marco de activación*, el compilador deberá asignar una región de la memoria para la creación de la *pila de marcos*. A este espacio se hace referencia mediante un registro llamado *apuntador de marco*, de modo que mientras se ejecuta una invocación de la función, se conoce dónde están almacenadas las variables locales, los parámetros de entrada y el valor devuelto.

Una ejecución a mano que muestra los estados de los marcos de activación se denomina *rastreo de activación* (Baase, 2002) y permite analizar el tiempo de ejecución de la función y comprender como funciona realmente la recursividad en el computador.

Cada invocación activa tiene un *marco de activación* único. Una invocación de función está activa desde el momento en que se entra en ella hasta que se sale, después de haber resuelto el problema que se le paso como parámetro. Todas las activaciones de la función recursiva que están activas simultáneamente tienen marcos de activación distintos. Cuando se sale de una invocación de función su marco de activación se libera automáticamente para que otras invocaciones puedan hacer uso de ese espacio y la ejecución se reanuda en el punto donde se hizo la invocación de la función.

Para ilustrar mejor estas ideas, en la figura 6.2 se presenta el *rastreo de activación* para la función factorial.

Una vez que la función haya encontrado el *criterio base*, es decir, un caso con solución no recursiva, éste marco devuelve a quien lo invocó el valor determinado para dicho criterio y libera el espacio ocupado; en este caso, el último marco retorna 1 para el marco anterior y desaparece. Y continúa un proceso regresivo en que cada marco de activación desarrolla sus operaciones utilizando el valor devuelto por

la función recursiva invocada y retorna el valor obtenido al marco que lo invocó.

Figura 6.2. Rastreo de activación para la función factorial

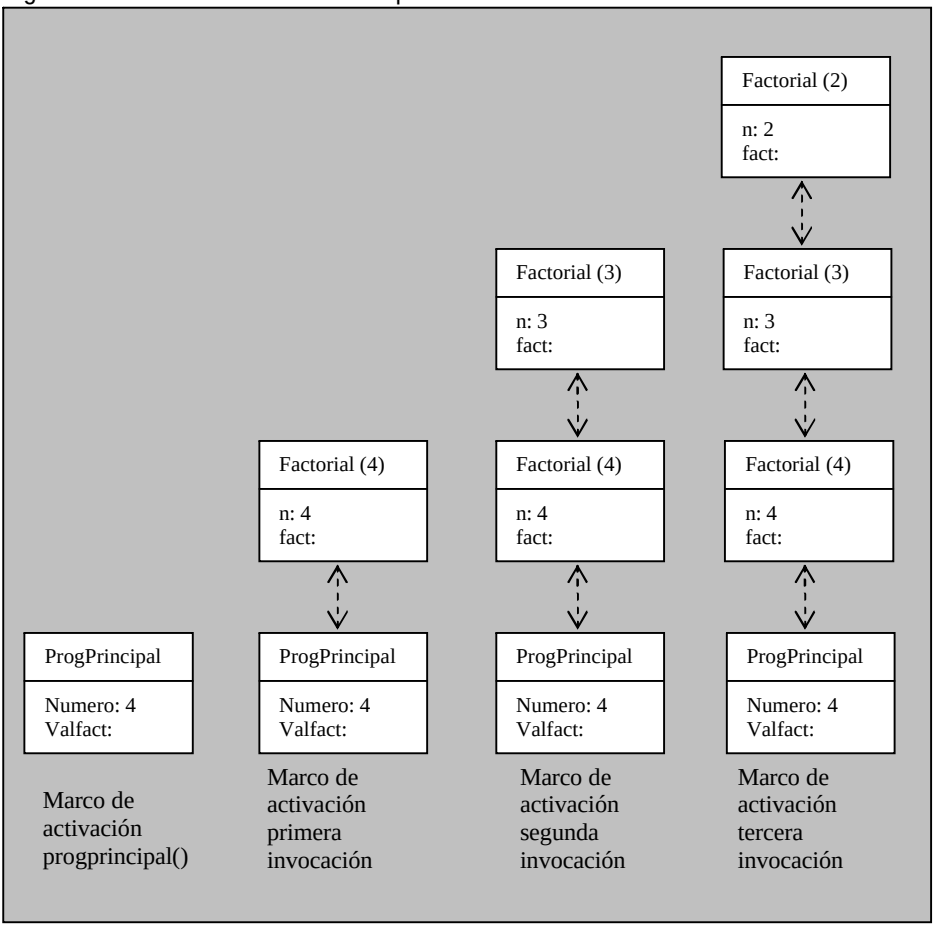
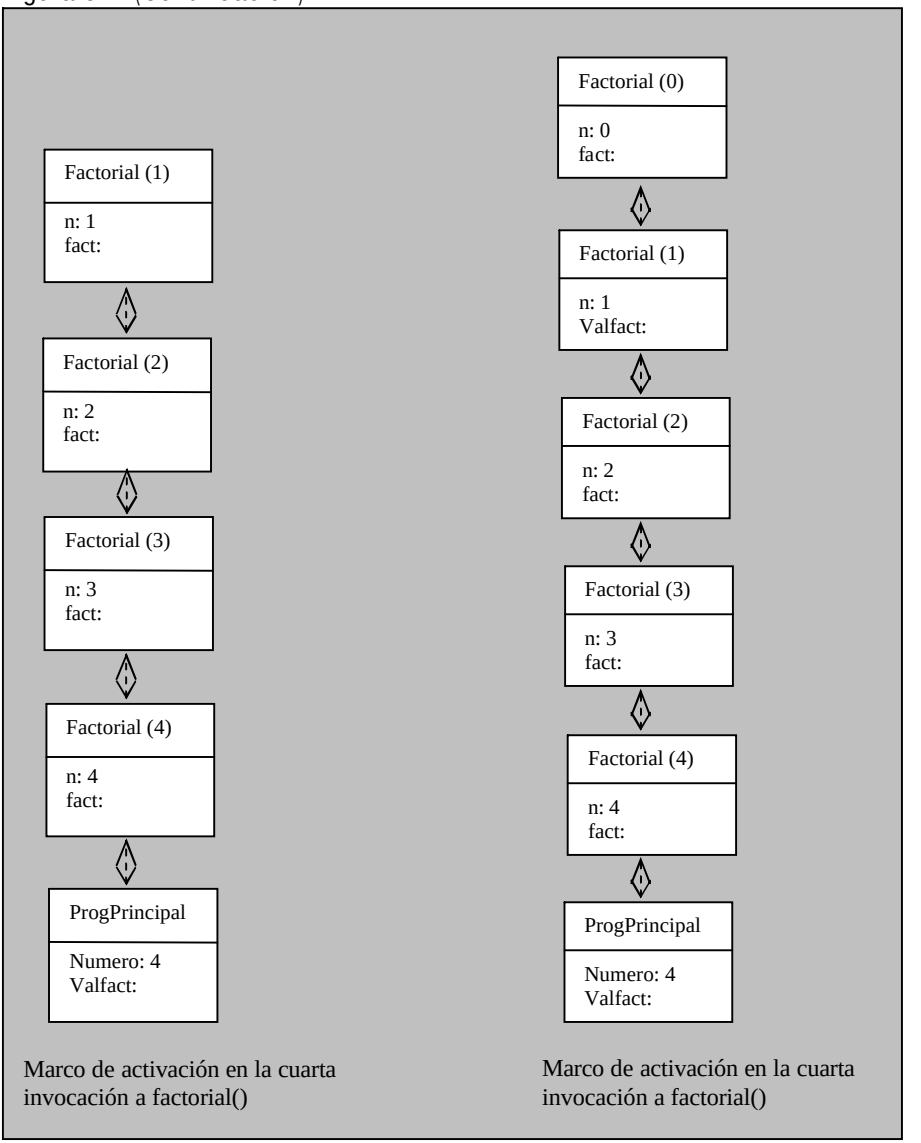


Figura 6.2. (Continuación)



Marco de activación en la cuarta invocación a factorial()

Marco de activación en la cuarta invocación a factorial()

6.4 ENVOLTURAS PARA FUNCIONES RECURSIVAS

Es común que las funciones recursivas se concentren en la búsqueda de un valor o en el desarrollo de una tarea concreta, como calcular el factorial, invertir una cadena, encontrar un valor en una estructura de datos u ordenar una lista y no se ocupen de otras tareas que no requieren ser recursivas como leer el dato a buscar, comprobar la validez de un argumento o imprimir los resultados.

Es decir, cuando se programa utilizando recursividad se encontrará que hay algunas tareas que no requieren un tratamiento recursivo y que pueden estar antes o después de la ejecución de la función recursiva. Para manejar estas tareas se definen programas, procedimientos o funciones *envoltura*.

Se denomina *envoltura* a los procedimientos no recursivos que se ejecutan antes y después de la invocación de una función recursiva y que se encargan de preparar los parámetros de la función y de procesar los resultados (Baase, 2002).

En el ejemplo del factorial se utiliza como envoltura a *ProgPrincipal*, que se encarga de leer el número que se utilizará como argumento de la función, invocar la función y mostrar el resultado. El seudocódigo se presenta en el cuadro 6.2. y el código en lenguaje Scheme en el cuadro 6.5.

Cuadro 6.2 Pseudo código del programa principal

```
1.   Programa ProgPrincipal
2.      Leer  num
3.      Imprimir "factorial de", num, " = ", factorial(num)
4.   Fin programa
```

6.5 TIPOS DE RECURSIVIDAD

La recursividad puede presentarse de diferentes maneras y dependiendo de ellas se han establecido al menos cuatro tipos diferentes de implementaciones recursivas.

Recursividad simple: se presenta cuando una función incluye un llamado a sí misma con un argumento diferente. Ejemplo de este tipo de recursividad es la función factorial().

Este tipo de recursividad se caracteriza porque puede pasarse fácilmente a una solución iterativa.

Recursividad múltiple: se da cuando el cuerpo de una función incluye más de una llamado a la misma función, por ejemplo, la función para calcular un valor de la serie Fibonacci.

Recursividad anidada: se dice que una función recursiva es anidada cuando entre los parámetros que se pasan a la función se incluye una invocación a la misma. Un ejemplo de recursividad anidada es la solución al problema de Ackerman.

Recursividad cruzada o indirecta: en este tipo de recursividad, el cuerpo de la función no contiene un llamado a sí misma, sino a otra función; pero, la segunda incluye un llamado a la primera. Puede ser que participen más de dos funciones. Este tipo de implementaciones también se conoce como *cadenas recursivas*. Como ejemplo de este tipo de recursividad se tiene la función para validar una expresión matemática.

El pseudocódigo y la implementación en lenguaje Scheme para los ejemplos mencionados se encuentran más adelante, en este mismo capítulo (secciones 6.6 y 6.7).

6.6 RECURSIVIDAD EN SCHEME

Como se ha explicado a lo largo de este capítulo, una función es recursiva cuando entre sus líneas cuenta con una o mas invocaciones a sí misma con al menos un argumento que cada vez se aproxima más al *criterio base*.

La ejecución de una función recursiva, en Scheme al igual que en muchos otros lenguajes, tiene dos fases:

a. Fase de activación. En la que se inician nuevas activaciones y se asignan recursos y
b. Fase de retorno. En la que la función libera los recursos utilizados y retorna el control al *marco de activación* que sigue en la pila de marcos (Coello, 1996).

Para implementar las funciones recursivas en Scheme se aplica los conceptos y técnicas que se han tratado en los capítulos anteriores. La estructura general se presenta en el cuadro 6.3.

Cuadro 6.3 Estructura de una función recursiva en Scheme

```
1.   (define nombreFuncion
2.     (lambda (parámetros)
3.       (Evaluación criterio Base)
4.          Retorno resultado criterioBase
5.          (Invocación recursiva)
6.        )
7.       )
8.   )
```

En la línea 1 se define un símbolo, que se identifica con nombreFunción, al cual se le asigna lo que aparece en la línea siguiente. En la línea 2, mediante la palabra ***lambda*** se especifica un procedimiento que utilizará los parámetros que recibe la función. En la línea 3 se evalúa si se ha alcanzado el *criterio base*, si la evaluación es verdadera se retorna el valor definido para dicho criterio (línea 4), en caso contrario se hace una nueva activación de la función (línea 5). Las líneas que siguen, simplemente, cierran los paréntesis que se

abrieron en las sentencias anteriores. La implementación de la función factorial se muestra en el cuadro 6.4.

Cuadro 6.4 Función para calcular el factorial de un número

```
1.  (define factorial
2.    (lambda (n)
3.      (if(= n 0)
4.         1
5.         (* n (factorial (- n 1)))
6.       )
7.      )
8.  )
```

```
> (factorial 5)
120
> (factorial 15)
1307674368000
```

Aunque la implementación completa se puede hacer en unas pocas líneas, se prefiere la presentación en varias líneas, una por cada sentencia y de igual manera para cada paréntesis buscando mayor claridad y legibilidad del código, y también, mayor similitud con la presentación en pseudocódigo. En los ejemplos que se desarrollan en la sección 6.7 se utilizará esta misma estrategia.

Normalmente, las funciones recursivas son activadas por primera vez por otra función o programa principal que las requiere, al que se conoce como *envoltura*. En el cuadro 6.5 se presenta el código Scheme de la *envoltura* para invocar la función factorial. En los ejemplos que se presentan más adelante, el código la envoltura se presentará junto con el de la función.

Cuadro 6.5 Envoltura de la función factorial

```
1.  (begin
2.    (define num (read))
3.    (display num)
4.    (display "! = ")
5.    (display (factorial num))
6.  )
```

6.7 EJEMPLOS DE SOLUCIONES RECURSIVAS

En esta sección se presentan algunos de los ejemplos comúnmente utilizados para explicar la recursividad. Además del código Scheme, se utiliza la notación en pseudocódigo buscando hacer más entendible el diseño de la solución recursiva y para que ésta se pueda implementar en cualquier lenguaje de programación que soporte recursividad.

6.7.1 Función sumatoria

Para calcular la sumatoria de los primeros *n* números enteros positivos, de la forma:

1 + 2 + 3 + 4 + 5 + 6 + … + (n-1) + n

Se define la función f_n donde *n* puede ser cualquier número mayor o igual a 1 ($n >= 1$).

Se conoce que al sumar 0 a cualquier número éste se mantiene. De ahí se desprende que la sumatoria de cero es cero. Para cualquier otro entero positivo, la sumatoria se calcula adicionando su propio valor a la sumatoria del número inmediatamente inferior. Así:

$f_0 = 0$
$f_1 = 1 + f_0$
$f_2 = 2 + f_1$
$f_3 = 3 + f_2$
$f_4 = 4 + f_3$
$f_n = n + f_{(n-1)}$

De esto se desprende que

a. Si $n = 0 \rightarrow f_n = 0$
b. Si $n > 0 \rightarrow f_n = n + f_{(n-1)}$

De esta manera se ha planteado una solución recursiva donde *a* representa el criterio base y *b* la invocación recursiva de la función.

El pseudocódigo de la función sumatoria se presenta en el cuadro 6.6.

Cuadro 6.6 Función sumatoria

```
1.   Función sumatoria(n)
2.       Si (n = 0) entonces
3.           Retornar 0
4.       Si no
5.           Retornar  (n + sumatoria(n – 1))
6.       Fin si
7.   Fin función
```

El código Scheme para la función sumatoria se presenta en el cuadro 6.7. Entre las líneas 1 y 8 se implementa la función recursiva, en la línea 4 y 5 se evalúa el criterio base y se devuelve el valor predeterminado, en la línea 5 se crea un nuevo marco de activación. A partir de la línea 10 se escribe el código de la *envoltura* que se encarga de leer el número, hacer la primera invocación de la función y luego presentar el resultado.

Cuadro 6.7 Programa recursivo para calcular sumatoria

```
1.   (define sumatoria
2.     (lambda (n)
3.       (if(= n 0)
4.          0
5.          (+ n (sumatoria (- n 1)))
6.        )
7.       )
8.   )
9.
10.  (begin
11.    (display “Numero: “)
12.    (define num (read))
13.    (display "Sumatoria(")
14.    (display num)
15.    (display ") = ")
16.    (display (sumatoria num))
17.    )
```

```
Numero: 10
Sumatoria(10) = 55
```

6.7.2 Función factorial

Se conoce como factorial de un número (*n*) al producto de los enteros comprendidos entre 1 y *n* inclusive, y se denota de la forma: *n!*

De manera que:

n! = 1 * 2 * 3 * 4 * … * (n-2) * (n - 1) * n

Se conoce que el factorial está definido para el caso de que *n* = 0, cuyo resultado es 1. Es decir, 0! = 1. A partir de esto se puede afirmar que está definido para cualquier entero positivo, así:

0! = 1	
1! = 1	1! = 1 * 0!
2! = 1 * 2 = 2	2! = 2 * 1!
3! = 1 * 2 * 3 = 3	3! = 3 * 2!
4! = 1 * 2 * 3 * 4 = 12	4! = 4 * 3!
5! = 5 * 4 * 3 * 2 * 1 = 60	5! = 5 * 4!

De esto se desprende que:

a. Si n = 0 → n! = 0
b. Si n > 0 → n! = n * (n – 1)!

Y esta es la definición correcta de una función recursiva, donde ***a*** es el criterio base, es decir, la solución conocida, y ***b*** es la invocación recursiva con un argumento más cercano a la solución. El pseudocódigo para esta función se presenta en el cuadro 6.1 y el código del programa en lenguaje Scheme en el cuadro 6.4.

6.7.3 Función potencia

Se define como cálculo de una potencia a la solución de una operación de la forma x^n, donde ***x*** es un número entero o real que se conoce como base y ***n*** es un entero no negativo conocido como exponente..

La potencia x^n es el producto de n veces x, de la forma:

$x^n = x * x * x * \ldots * x$ (n veces x)

Por las propiedades de la potenciación se tiene que cualquier número elevado a 0 da como resultado 1 y que cualquier número elevado a 1 es el mismo número.

$x^0 = 1$
$x^1 = x$

Para este ejercicio se extiende la primera propiedad, también, para el caso de x = 0. Aunque normalmente se dice que este resultado no está definido, para explicar recursividad esto es irrelevante.

Para proponer una solución recursiva es necesario considerar la potencia x^n en función de una expresión más cercana a la identidad $x^0 = 1$.

Se propone el siguiente ejemplo:

$2^0 = 1$	
$2^1 = 2$	$2^1 = 2 * 2^0 = 2$
$2^2 = 4$	$2^2 = 2 * 2^1 = 2 * 2 = 4$
$2^3 = 8$	$2^3 = 2 * 2^2 = 2 * 4 = 8$
$2^4 = 16$	$2^4 = 2 * 2^3 = 2 * 8 = 16$
$2^5 = 32$	$2^5 = 2 * 2^4 = 2 * 16 = 32$

Como se aprecia en la columna de la derecha, cada potencia puede plantearse utilizando la solución de la potencia anterior, con exponente menor.

En consecuencia:

a. Si $n = 0 \rightarrow x^n = 1$
b. Si $n > 0 \rightarrow x^n = x * x^{n-1}$

Donde ***a*** es el criterio base y ***b*** es la invocación recursiva.

En el cuadro 6.8 se presenta el pseudocódigo para calcular recursivamente cualquier potencia de cualquier número.

Cuadro 6.8 Función potencia

```
1.  Función potencia(x, n)
2.     Real: pot
3.     If(n = 0)
4.        Retornar 1
5.     Si no
6.        Retornar (x * potencia(x, n-1))
7.     Fin si
8.  Fin función
```

En las líneas 3 y 4 se soluciona el problema cuando se presenta el criterio base, en caso contrario, se debe hacer una invocación recursiva de la función, según las líneas 5 y 6.

La implementación de esta función en lenguaje Scheme se muestra en el cuadro 6.9.

Cuadro 6.9 Programa para calcular una potencia

```
1.   (define potencia
2.     (lambda (x n)
3.       (if(= n 0)
4.          1
5.          (* x (potencia x (- n 1)))
6.       )
7.      )
8.    )
9.
10.  (begin
11.    (display "\n Base: ")
12.    (define base (read))
13.    (display "\n Exponente: ")
14.    (define expo (read))
15.    (display base)
16.    (display " ^ ")
17.    (display expo)
18.    (display " = ")
19.    (display (potencia base expo))
20.  )
```

Cuadro 6.9 (Continuación)

```
Base: 2
Exponente: 4
2 ^ 4 = 16

Base: 5
Exponente: 4
5 ^ 4 = 625
```

6.7.4 Función fibonacci

La serie Fibonacci tiene muchas aplicaciones en ciencias de la computación, en matemáticas y en teoría de juegos. Fue publicada por primera vez en 1202 por un matemático italiano del mismo nombre, en su libro titulado *Liberabaci* (Brassard, 1997).

Fibonacci planteó el problema de la siguiente manera: supóngase que una pareja de conejos da lugar a dos descendientes cada mes y cada nuevo ejemplar comienza su reproducción después de dos meses. De manera que al comprar una pareja de conejos, en los meses uno y dos se tendrá una pareja, pero al tercer mes se habrán reproducido y se contará con dos parejas, en el mes cuatro solo se reproduce la primera pareja, así que el número aumentará a tres parejas; en el mes cinco, comienza la reproducción de la segunda pareja, con lo cual se obtendrán cinco parejas, y así sucesivamente.
Si ningún ejemplar muere, el número de conejos que se tendrán cada mes está dado por la sucesión: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …, que corresponde a la relación:

Mes (n)	0	1	2	3	4	5	6	7	8	9	..
Pares de conejos (f_n)	0	1	1	2	3	5	8	13	21	34	..

Cuya representación formal es:

a. $f_0 = 0$
b. $f_1 = 1$
c. $f_n = f_{n-1} + f_{n-2}$

Es decir, en el mes 0, aun no se tiene ningún par de conejos; en el mes uno, se adquiere la primera pareja; y, en el mes dos, se mantiene la misma pareja comprada, porque aun no comienzan a reproducirse. Pero a partir del mes tres hay que calcular la cantidad de parejas que se tienen considerando la condición de que cada par genera un nuevo par cada mes, pero solo comienza a reproducirse después de dos meses.

De esta manera, el problema está resuelto para el mes cero y para el mes uno, estos se toman como criterio base (*a*, *b*) y a partir del segundo mes se debe tener en cuenta los meses anteriores. Esto genera una invocación recursiva de la forma $f_n = f_{n-1} + f_{n-2}$ para $n >= 2$.

En el pseudocódigo que se presenta en el cuadro 6.10 para la función que genera la serie Fibonacci se aprecia que cada activación tiene definidos dos criterios bases (líneas 3 y 6) y, si aun no se ha llegado a estos, en cada marco de activación se generan dos nuevas invocaciones recursivas (línea 9).

Cuadro 6.10 Pseudocódigo de la función Fibonacci

```
función Fibonacci(n)
1.   Inicio
2.     Si (n = 0) entonces
3.        Retornar 0
4.     Si no
5.        Si (n = 1) entonces
6.           Retornar 1
7.        Si no
8.           Retornar (Fibonacci(n – 1) + Fibonacci(n – 2))
9.        Fin si
10.    Fin si
11.  Fin función
```

En el cuadro 6.11 se muestra el código del programa en Scheme para calcular el valor de la serie Fibonacci para un número ***n*** ingresado por el usuario. Obsérvese que en la línea 7 se hace dos invocaciones a la función recursiva, de esta manera los marcos de activación forman estructura de árbol binario recursivo.

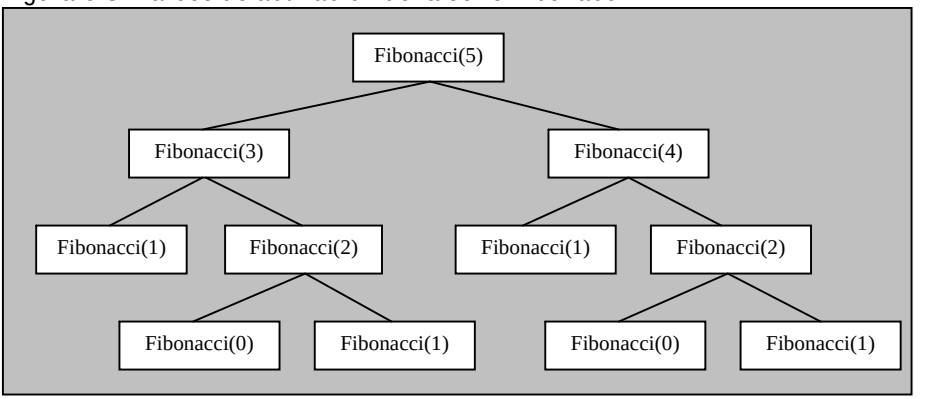
Cuadro 6.11 Programa para calcular la serie Fibonacci

```
1.   (define Fibonacci
2.     (lambda (n)
3.       (if (= n 0)
4.           0
5.           (if (= n 1)
6.               1
7.               (+ (Fibonacci(- n 2)) (Fibonacci(- n 1)))
8.            )
9.        )
10.      )
11.    )
12.
13.  (begin
14.    (display "Número: ")
15.    (display (Fibonacci (read)))
16.    )
```

```
Numero: 20
6765
```

En la figura 6.3 se presenta el árbol que se forma al calcular el valor de la serie para el número 5.

Figura 6.3 Marcos de activación de la serie Fibonacci



6.7.5 Máximo Común Divisor

El Máximo Común Divisor (MCD) entre dos o más números es el mayor de los divisores comunes. Se conocen dos métodos para encontrar el MCD entre dos números: el primero consiste en descomponer cada número en sus factores primos, escribir los números en forma de potencia y luego tomar los factores comunes con menor exponente y el producto de estos será el MCD; segundo es la búsqueda recursiva en la que se verifica si el segundo número es diviso del primero, en cuyo caso será el MCD, si no lo es se prueba con el módulo y así sucesivamente con un número cada vez más pequeño que tiende a 1 como el divisor común a todos los números. Este último se conoce como algoritmo de Euclides.

Por ejemplo, si se desea encontrar el MCD de 24 y 18, aplicando el algoritmo de Euclides, se divide el primero sobre el segundo (24/18= 1, residuo = 6), si la división es exacta se tiene que el MCD es el segundo número, pero si no lo es, como en este caso, se vuelve a buscar el MCD entre el segundo número (18) y el módulo de la división (6) y de esa manera hasta que se lo encuentre. Si los dos números son primos se llegará hasta 1.
Si $f(a,b)$ es la función que devuelve el Máximo Común Divisor de a y b, entonces se tiene que:

$c = a \bmod b$

a. Si $c = 0 \rightarrow f(a,b) = b$
b. Si $c > 0 \rightarrow f(a,b) = f(b,c)$

El código Scheme de la función MCD se presenta en el cuadro 6.12.

6.8 EFICIENCIA DE LA RECURSIVIDAD

En general, una versión iterativa se ejecutará con más eficiencia en términos de tiempo y espacio que una versión recursiva. Esto se debe a que, en la versión iterativa, se evita la información general implícita

al entrar y salir de una función. En la versión recursiva, con frecuencia, es necesario apilar y desapilar variables anónimas en cada campo de activación. Esto no es necesario en el caso de una implementación iterativa donde los resultados se van reemplazando en los mismos espacios de memoria.

Cuadro 6.12. Función para encontrar el Máximo Común Divisor

```
1.  (define mcd
2.    (λ(a b)
3.      (define c (remainder a b))
4.      (if(= c 0)
5.         B
6.         (mcd b c)
7.       )
8.     )
9.  )
```

```
> (mcd 18 9)
9
> (mcd 30 24)
6
> (mcd 215 120)
5
```

Por otra parte, la recursividad es la forma más natural de solucionar algunos tipos problemas, como es el caso de: ordenamiento rápido (QuickSort), las torres de Hanoy, las ocho reinas, la conversión de prefijo a posfijo o el recorrido de árboles, que aunque pueden implementarse soluciones iterativas, la solución recursiva surge directamente de la definición del problema.

El optar por métodos recursivos o iterativos es, más bien, un conflicto entre la eficiencia de la máquina y la eficiencia del programador (Langsam, 1997). Considerando que el costo de la programación tiende a aumentar y el costo de computación a disminuir, no vale la pena que un programador invierta tiempo y esfuerzo desarrollando una complicada solución iterativa para un problema que tiene una solución recursiva sencilla, como tampoco vale la pena implementar soluciones recursivas para problemas que pueden solucionarse fácilmente de manera iterativa. Muchos de los ejemplos de soluciones recursivas de este capítulo se presentan con fines didácticos más no por su eficiencia.

Conviene tener presente que la demanda de tiempo y espacio extra, en las soluciones recursivas, proviene principalmente de la creación de los espacios de activación de las funciones y del apilamiento de resultados parciales, de manera que éstas pueden optimizarse reduciendo el uso de variables locales. A la vez que las soluciones iterativas que utilizan pilas pueden requerir tanto tiempo y espacio como las recursivas.

CAPÍTULO 7

ARREGLOS

"Quien no se resuelve
a cultivar el hábito de pensar,
se pierde el mayor placer de la vida".
T. A.Edison.

En términos generales un vector es un conjunto de variables que se diferencian entre sí por la posición que ocupan dentro del vector.

Los vectores son muy útiles porque permiten agrupar datos y manipularlos fácilmente. Scheme, a diferencia de la mayoría de lenguajes, admite elementos de diferente tipo, permitiendo de esta manera un manejo más eficiente de la información.

Para manejar vectores es preciso contar con un conjunto de funciones que permitan realizar tareas como: definir, crear, actualizar y consultar un vector. Scheme ofrece varias funciones para el trabajo con vectores, y el programador puede implementar algunas otras que le sean necesarias; de esta manera, es muy fácil aplicar este tipo de datos.

Este capítulo presenta los conceptos importantes sobre el tema, las funciones que Scheme ofrece para la utilización de vectores y varios ejemplos de aplicación.

7.1 EL CONCEPTO DE VECTOR

Un vector es una colección finita y ordenada de elementos que reciben un nombre común.

Se dice que es:

Finita: porque todo vector tiene un límite; es decir, debe determinarse cuál será el número máximo de elementos que podrán formar parte de él.

Ordenada: porque se puede determinar cuál es el primer elemento, el segundo, el tercero,.... y el n-ésimo elemento.

En cada posición del vector se puede almacenar un valor de cualquier tipo de datos, entre ellos cadenas, caracteres, números enteros, reales, etc. Para referirse a un determinado elemento de un vector se utiliza un índice, que especifique su posición relativa en él.

7.2 DEFINICIÓN DE UN VECTOR EN SCHEME

Para manejar un vector en Scheme es necesario definirlo, en este proceso se le asigna un nombre, una dimensión y opcionalmente el contenido de cada posición.

El formato general para definir un vector es el siguiente:

```
(define nombre-vector  #(obj…..))
```

Donde nombre-vector es el nombre que el programador le da al vector según su gusto y conveniencia; el caracter # indica que se trata de un vector y obj…. es el elemento o conjunto de elementos que contiene el vector. Los elementos del vector pueden ser del mismo o de diferentes tipos de datos, por ejemplo, se puede utilizar un vector con una colección de números o un vector que contenga

posiciones con valores numéricos junto con otras posiciones con cadenas, caracteres, símbolos, listas, etc.

El tamaño del vector está dado por el número de elementos que asignaron en la definición; para acceder a cada elemento se utiliza un índice, el cual es un número entero entre 0 y el valor dado por la longitud del vector, cuando el índice es 0 hace referencia al primer elemento, 1 para el segundo elemento y así consecutivamente para cada uno de los elementos del vector.

Algunos ejemplos de vectores son:

a.) Vector de nombres de estudiantes:

La expresión:

```
(define estudiantes #(Julián Luis Ana Lucía Pedro Lilian Sofía))
```

Crea un vector de 7 posiciones con un índice que va de 0 a 6:

Vector estudiantes

Julián	Luis	Ana	Lucia	Pedro	Lilian	Sofía
0	1	2	3	4	5	6

Observe que el índice de los elementos del vector es un número que inicia en 0 y se incrementa en 1 hasta el último elemento.

b.) Vector notas de estudiantes:

La expresión:

```
(define notas #(01 Julián  4  4.5  5))
```

Crea un vector de 5 posiciones con valores de diferentes tipos de datos, en la posición 0 se almacena el código del estudiante, en la posición 1 el nombre, en las posiciones 2, 3 y 4 se almacenan las notas del primero, segundo parcial y examen final.

Vector notas

01	Julián	4	4.5	5
0	1	2	3	4

El contenido de este vector tiene diferentes tipos de datos: entero para el código, string para el nombre y real para las notas parciales.

c.) Vector de números racionales:

La instrucción:

```
(define racionales #(3/5  5/4  2/6  3/10))
```

Crea un vector de 4 posiciones con valores racionales, en la posición 0 se almacena el número 3/5, en la posición 1 5/4, en la posición 2 2/6 y en la posición 3 3/10:

Vector racionales

3/5	5/4	2/6	3/10
0	1	2	3

Ejemplo 7.1. Programa en Scheme para crear vectores de diferentes tipos y mostrar su contenido. El cuadro 8.1. presenta el código de este programa.

Cuadro 7.1 Definición de vectores

```
1.    (begin
2.     ;Vector de nombres
3.     (define estudiantes #(Julián Luis Ana Lucía Pedro
    Lilian Sofía))
4.     (display estudiantes)
5.     (newline)
6.
7.     ;Vector de tipos de datos diferentes
8.     (define notas # (01 Julián  4  4.5  5))
```

Cuadro 7.1 (Continuación)

```
9.      (display notas)
10.     (newline)
11.
12.     ;Vector de números racionales
13.     (define racionales # (3/5  5/4  2/6  3/10))
14.     (display racionales)
15.     (newline)
16.
17.     ;Vector de caracteres
18.     (define vocales #(a b c d e))
19.     (display vocales)
20.     (newline)
21.
22.     ;Vectores con listas y otros tipos de datos
23.     (define mixta #(0 (2 2 2) "Jenny")
24.     (display mixta)
25.     (newline)
26.
27.     ;Vector de números reales
28.     (define num-reales #(3.5 4.89 12.34))
29.     (display num-reales)
30.     (newline)
```

```
31.     #(Julián Luis Ana Lucía Pedro Lilian Sofía)
32.     #(1 Julián 4 4.5 5)
33.     #(3/5 5/4 1/3 3/10)
34.     #(a b c d e)
35.     #(0 (2 2 2) Jenny)
36.     #(3.5  4.89  12.34)
```

7.3 FUNCIONES PARA MANEJO DE VECTORES

El lenguaje de programación Scheme tiene diferentes funciones que permiten manipular un vector, mediante ellas se puede crear un vector, acceder a una posición del vector, calcular su tamaño, modificar su contenido, etc. Las funciones para manejar vectores son tareas ya predefinidas por el lenguaje Scheme y pueden ser utilizadas en cualquier parte del programa; entre las más importantes se encuentran las relacionadas en el cuadro 7.2.

Cuadro 7.2 Funciones para manejo de vectores

	Función	**Descripción**
1.	Make-vector	Retorna un vector de n posiciones inicializadas con un mismo valor.
2.	Vector	Retorna un vector compuesto por los parámetros enviados en su llamado.
3.	list-vector	Crea un vector inicializado con los elementos que contenga una lista.
4.	vector-length	Retorna el número de posiciones que contiene el vector.
5.	vector-ref	Retorna el contenido de una posición del vector.
6.	vector-set!	Almacena un valor en una posición del vector.
7.	vector-fill!	Almacena un valor en cada posición del vector.
8.	vector?	Determina si un objeto es un vector.

7.3.1 *make-vector*

La función *make-vector* retorna un vector de *n* posiciones inicializadas con un mismo valor.

La sintaxis de esta función es:

```
(make-vector k)
```

```
(make-vector k obj)
```

Esta función tiene dos formatos, el primero retorna un vector de k posiciones inicializadas en 0, el segundo retorna un vector de k posiciones inicializadas con un valor obj de cualquier tipo, por ejemplo un carácter, una cadena, un número, etc.

Esta función se la puede combinar con define para referenciar al vector por medio de una variable, además permite darle un tamaño por medio de un número e inicializar sus posiciones con un valor cualquiera que considere el programador, de acuerdo con las necesidades del programa que se esté planteando.

Por ejemplo, si se quiere crear un vector de 10 posiciones para almacenar números enteros se lo puede hacer así:

```
(define enteros (make-vector 10))
```

Vector enteros

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

Se podría crear el vector inicializado con un número diferente de 0, por ejemplo con el valor 23, de la siguiente manera:

```
(define enteros (make-vector 10 23))
```

Vector enteros

23	23	23	23	23	23	23	23	23	23
0	1	2	3	4	5	6	7	8	9

Igualmente, se puede crear un vector con elementos de otros tipos de datos, por ejemplo, un vector de 8 posiciones inicializadas con el carácter s.

```
(define caracteres (make-vector 8  #\s))
```

Vector caracteres

S	s	s	S	s	s	s	s
0	1	2	3	4	5	6	7

Ejemplo 7.2. Programa para crear vectores con define y make-vector y mostrar el contenido de cada uno de ellos utilizando la función display. En el cuadro 7.3 se muestra la solución.

Cuadro 7.3 Creación de vectores con la función *make-vector*

```
 1.  (begin
 2.    (define enteros (make-vector 10))
 3.    (define v2 (make-vector 10 39))
 4.    (define v3 (make-vector 6 #\s))
 5.    (define v4 (make-vector 5  null))
 6.    (define v5 (make-vector 12  0.0))
 7.    (display enteros)
 8.    (newline)
 9.    (display v2)
10.    (newline)
11.    (display v3)
12.    (newline)
13.    (display v4)
14.    (newline)
15.    (display v5)
16.  )
```

```
#(0 0 0 0 0 0 0 0 0 0)
#(39 39 39 39 39 39 39 39 39 39)
#(s s s s s s)
#(() () () () ())
#(0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0)
```

7.3.2 *vector*

La función *vector* retorna un vector inicializado con los elementos que se le envían como parámetros.

La sintaxis de esta función es:

```
(vector obj1 [obj2…objn])
```

Donde obj1 [obj2….objn] es el o los parámetros con los cuales se va inicializar el vector, cada uno de los parámetros pueden ser del mismo o de diferentes tipos de datos.

Esta función también se puede combinar con define para asignar el valor retornado por la función a una variable, por ejemplo, se va a crear un vector que guarde la fecha de nacimiento de una persona, (2 de Junio de 2006), una vez creado se le asignará el nombre fecha con define:

```
(define fecha (vector  2  "Junio"  1996))
```

Vector fecha

2	Junio	1996
0	1	2

Ejemplo 7.3. Crear vectores utilizando las funciones define y vector y mostrar su contenido. El código se presenta en el cuadro 7.4.

Cuadro 7.4. Creación de vectores con la función *vector*

```
 1. (begin
 2.    (define fecha (vector 2 "Junio" 1996))
 3.    (define semestre (vector "Ene" "Feb" "Mar" "Abr"
         "May" "Jun"))
 4.    (define celulares (vector 1 "Nokia" 2 "Ericcson" 3
         "Samsung"))
 5.    (display fecha)
 6.    (newline)
 7.    (display semestre)
 8.    (newline)
 9.    (display celulares)
10.    (newline)
11. )
```

```
#(2 Junio 1996)
#(Ene Feb Mar Abr May Jun)
#(1 Nokia 2 Ericcson 3 Samsung)
```

7.3.3 *list-vector*

Crea un vector inicializado con los elementos que compone una lista. La sintaxis de esta función es:

```
(list->vector  lista)
```

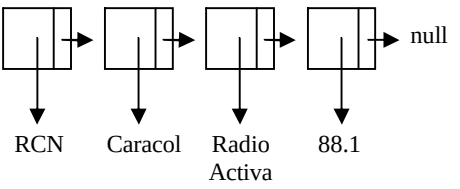
Donde lista es el nombre de una lista o una lista de la forma '(obj1 [obj2….objn]).

Por ejemplo si se tiene una lista de emisoras compuesta por los nombres de emisoras de radio, es posible, pasar esa información a un vector, de la siguiente manera:

Primero se define la lista de emisoras denominada lst-emisoras:

```
(define lst-emisoras '("RCN" "Caracol"
"RadioActiva"  "88.1"))
```

Lista lst-emisoras



Posteriormente se crea el vector v-emisoras a partir de la lista definida anteriormente así:

```
(define v-emisoras (list-vector  lst-emisoras))
```

Vector emisoras

RCN	Caracol	RadioActiva	88.1
0	1	2	3

El tema de listas se lo trata en el siguiente capítulo, se sugiere referirse a él para mayor comprensión de este tema.

7.3.4 *vector-length*

La función vector-length retorna el número de posiciones que tiene un vector. Su sintaxis es.

```
(vector-length vector)
```

Si se tiene el vector *imp* compuesto por los números impares entre 0 y 9, el valor que retorna la función vector-length es 5.

imp

1	3	5	7	9
0	1	2	3	4

> Observe que el índice llega hasta 4 porque inicia en 0, pero la longitud del vector es 5 porque está compuesto por 5 elementos.

Para obtener el total de elementos del vector *imp* se debe escribir:

```
(vector-length imp)
```

Esta expresión puede utilizarse de diferentes formas, por ejemplo:

a. Se puede asignar su retorno a una variable

```
(define num-elementos (vector-length imp))
```

En este caso el valor 5 es asignado a la variable num-elementos.

b. Se puede utilizar en una condición o en un ciclo

```
(if (< (vector-length imp) 3)
   (display "El vector tiene menos de tres
             elementos")
   (display "El vector tiene 3 o más elementos")
)
```

En este ejemplo se pregunta si la longitud del vector imp es menor que 3, por verdadero muestra por pantalla: "El vector tiene menos de tres elementos", por falso muestra: "El vector tiene 3 o más elementos".

c. Se puede utilizar para mostrar directamente el total de elementos de un vector:

```
(display (vector-length  imp))
```

Esta línea muestra en pantalla el valor 5.

d. Se puede utilizar como parámetro en el llamado a una función o procedimiento:

```
(func (vector-length imp))
```

Esta expresión consiste en llamar a la función func, enviándole como parámetro la longitud de imp.

Puede utilizarse de otras formas dependiendo de lo que se necesite hacer, sin embargo, lo más importante es tener en claro que la función vector-length retorna un valor entero.

7.3.5 *vector-ref*

La función vector-ref retorna el contenido de una posición del vector. El valor retornado se puede utilizar para cualquier tipo de operación, por ejemplo, mostrarlo en pantalla, realizar una operación matemática, utilizarlo en una condición, enviarlo como un parámetro de un procedimiento, etc. En otras palabras, cada elemento del vector por separado cumple la misma función que una variable independiente.

La sintaxis de esta función es:

```
(vector-ref  vector  k)
```

Donde k es la posición del vector del cual se obtiene su contenido.

Por ejemplo, se puede almacenar en un vector una colección de instituciones de educación superior, de la siguiente forma:

```
(define universidades #(Autónoma Cooperativa I.U.Cesmag Mariana
Nariño UNAD))
```

universidades

Autónoma	Cooperativa	I.U.Cesmag	Mariana	Nariño	UNAD
0	1	2	3	4	5

Si se quiere mostrar todas las universidades se puede utilizar la función display así:

```
(display universidades)
```

Esta línea muestra:

```
#(Autónoma Cooperativa I.U.Cesmag Mariana
Nariño UNAD)
```

Pero, si lo que se quiere es mostrar solo uno de los elementos, por ejemplo Mariana, se debe mostrar la posición 3 del vector de la siguiente manera:

```
(display (vector-ref universidades 3))
```

Esta línea muestra:

```
Mariana
```

Para mostrar todos los elementos del vector, accediendo uno a uno a sus elementos, se puede proceder de tres formas:

1. Hacer tantos display como posiciones tenga el vector,
2. Hacer un ciclo que en cada interacción imprima una posición diferente

3. Hacer un procedimiento recursivo que en cada llamado muestre un elemento del vector.

Ejemplo 7.4. Programa para mostrar el contenido de un vector elemento por elemento.

En el programa del cuadro 7.5 se presenta la primera forma. Consiste en utilizar tantos display como elementos tenga el vector, para este caso, se necesitan seis display para las posiciones 0 a la 5 del vector universidades:

Cuadro 7.5 Programa para mostrar un vector posición por posición

```
 1.  (begin
 2.    (define universidades #(Autónoma Cooperativa
                    I.U.Cesmag Mariana  Nariño UNAD))
 3.    (display (vector-ref universidades 0))
 4.    (newline)
 5.    (display (vector-ref universidades 1))
 6.    (newline)
 7.    (display (vector-ref universidades 2))
 8.    (newline)
 9.    (display (vector-ref universidades 3))
10.    (newline)
11.    (display (vector-ref universidades 4))
12.    (newline)
13.    (display (vector-ref universidades 5))
14.  )
```

```
Autónoma
Cooperativa
I.U.Cesmag
Mariana
Nariño
UNAD
```

Ejemplo 7.5. Programa para mostrar el contenido de un vector utilizando un ciclo do.

En el programa del cuadro 7.6 se implementa un ciclo con la variable i desde 0 hasta que sea igual o mayor que la cantidad de

elementos `(>= i (vector-length universidades))` y en cada ocasión se muestra el contenido del vector en la posición i.

Cuadro 7.6. Programa para mostrar un vector utilizando un ciclo

```
1.  (begin
      (define universidades #(Autónoma Cooperativa
2.                            I.U.Cesmag Mariana  Nariño
                              UNAD))
3.    ; Ciclo para mostrar los datos del vector
      (do ((i 0 (+ i 1))) ((>= i (vector-length
4.                                  universidades)))
5.      (display (vector-ref universidades i))
6.      (newline)
7.    ) ; Fin do
8.  ) ; Fin begin
```

```
Autónoma
Cooperativa
I.U.Cesmag
Mariana
Nariño
UNAD
```

Ejemplo 7.6. Programa para mostrar el contenido de un vector aplicando recursividad.

En el cuadro 7.7 se implementa la tercera alternativa. Consiste en utilizar un procedimiento recursivo, de tal manera que cada vez que se lo llame muestra en pantalla una posición del vector, para ello se le envía la posición que se quiere mostrar (i) y hace el llamado así mismo enviándole la próxima posición (+ i 1), este proceso se va a repetir mientras la posición recibida sea menor que la longitud del vector, de lo contrario no se hace llamado recursivo y el control retorna hasta llegar al primer llamado en el bloque principal:

Cuadro 7.7. Programa para mostrar un vector recursivamente

```
1.  (define muestra-vector
2.    (λ (universidades i)
3.      (if (< i  (vector-length universidades))
4.       (begin
5.          (display (vector-ref universidades i))
6.          (newline)
```

Cuadro 7.7 (Continuación)

```
 7.          (muestra-vector universidades (+ i 1))
 8.        ) ; Fin begin
 9.       ) ; Fin if
10.    ) ; Fin lambda
11.  ) ; Fin función
12.
13.  ; Bloque principal
14.  (begin
       (define universidades #(Autónoma Cooperativa
15.                          I.U.Cesmag Mariana  Nariño
                             UNAD))
16.    (muestra-vector universidades 0)
17.  ) ; Fin bloque principal
```

```
Autónoma
Cooperativa
I.U.Cesmag
Mariana
Nariño
UNAD
```

Ejemplo 7.7. Mostrar el contenido de un vector en orden inverso.

En el cuadro 7.8 se utiliza un ciclo que inicializa la variable i en la última posición del vector y con decrementos sucesivos va mostrando cada una de las posiciones hasta llegar a la posición 0. La sentencia principal en este programa es la cabecera del ciclo:

```
(do ((i (- (vector-length a) 1) (- i 1))) ((< i 0))
```

A la variable i se le asigna la longitud del vector menos 1

Por cada iteración, la variable i se disminuye en 1

El ciclo se ejecuta hasta que i sea menor que 0

A la variable i se le asigna la longitud del vector menos 1 y el ciclo inicia su proceso, en cada iteración muestra el contenido del vector

en la posición i, luego disminuye en uno la variable i mediante la sentencia (- i 1), posteriormente el ciclo se detiene cuando i es menor que 0, ((< i 0)) y finalmente sale del ciclo y termina.

La función vector-ref es utilizada para retornar el contenido de la posición i del vector a mediante la instrucción:

```
(display (vector-ref a i))
```

Cuadro 7.8. Programa para mostrar un vector en orden inverso

```
 1.  (begin
 2.    (define a #(12 14 16 18 20))
 3.    (do ((i (- (vector-length a) 1) (- i 1))) ((< i 0))
 4.      (display "Contenido en la posición ")
 5.      (display i)
 6.      (display " : ")
 7.      (display (vector-ref a i))
 8.      (newline)
 9.    ); Fin do
10.  ); Fin begin
```

```
Contenido en la posición 4 : 20
Contenido en la posición 3 : 18
Contenido en la posición 2 : 16
Contenido en la posición 1 : 14
Contenido en la posición 0 : 12
```

7.3.6 *vector-set!*

La función vector-set! almacena un valor cualquiera en una de las posiciones del vector. Su sintaxis es:

```
(vector-set!  vector  k  obj)
```

Donde k es la posición del vector donde se va a almacenar el valor obj.

La función vector-set! es muy útil cuando se tiene un vector que debe actualizar sus datos, por ejemplo, la cantidad de elementos de un inventario, los precios de cada uno de los productos que comercializa una empresa, las ventas de esos productos, etc.

Por ejemplo, se puede almacenar en un vector los precios de diferentes productos así:

```
(define precios (vector 45000 75000 185000 45000
25000))
```

precios

45000	75000	185000	45000	25000
0	1	2	3	4

Como los precios se modifican constantemente, es posible que se desee cambiar alguno de ellos, para lograrlo, es necesario utilizar la función vector-set!, por ejemplo, para modificar el valor 185000 almacenado en la posición 2 del vector por 200000, en esta caso la instrucción es:

```
(vector-set! precios  2  200000)
```

Con lo cual el vector queda así:

precios

45000	75000	200000	45000	25000
0	1	2	3	4

También se puede modificar el valor de una posición por el valor digitado por el usuario en tiempo de ejecución, para ello, se utiliza la función read de la siguiente manera:

```
(vector-set! precios 2 (read))
```

Ejemplo 7.8. Programa que crea un vector y lo llena con datos capturados por el teclado.

En el programa del cuadro 7.9 se implementa un ciclo que itera la función read para capturar valores y vector-set! para almacenarlos en cada posición del vector.

Cuadro 7.9. Programa que captura datos y los guarda en un vector

```
 1.  (begin
 2.    (define v(make-vector 4))
 3.    (do ((i 0 (+ i 1))) ((>= i (vector-length v)))
 4.      (display "Digite valor para la posición ")
 5.      (display i)
 6.      (display " : ")
 7.      (vector-set! v i (read));
 8.      ); Fin do
 9.    (display "En el vector se almacenaron los
                siguientes valores: ")
10.    (display v); Muestra todos los elementos del vector
11.  ); Fin begin
```

```
Digite valor para la posición 0 : 16
Digite valor para la posición 1 : 13
Digite valor para la posición 2 : 9
Digite valor para la posición 3 : 23
En el vector se almacenaron los siguientes valores: #(16
13 9 23)
```

Ejemplo 7.9. Un programa que lee dígitos y los guarda en un vector, luego cuenta cuántas veces aparece cada dígito.

En este ejemplo se utilizan dos vectores, el primero guarda los dígitos y el segundo el contador para cada dígito. En cada posición del primer vector puede haber un número entre 0 y 9, por esto el segundo vector debe tener 10 posiciones, en la primera registrará las veces que está el 0, en la segunda las del 1, y así sucesivamente hasta la posición 10 que registrará las apariciones del 9. Por ejemplo, si en el vector dígitos, en una de sus posiciones, se encuentra el valor 4, entonces la posición cuatro del vector contador se incrementará en uno.

Observemos el siguiente caso donde el vector dígitos se conforma por:

dígitos

3	5	3	0	6	4	5	5
0	1	2	3	4	5	6	7

Por cada posición del vector dígitos, el vector contador se actualiza aumentando en uno el contenido de una de sus casillas de acuerdo al número encontrado en dígitos, a continuación se observa este proceso.

Vector dígitos		Vector contador									
Posición	Contenido	0	1	2	3	4	5	6	7	8	9
0	3	0	0	0	1	0	0	0	0	0	0
1	5	0	0	0	1	0	1	0	0	0	0
2	3	0	0	0	2	0	1	0	0	0	0
3	0	1	0	0	2	0	1	0	0	0	0
4	6	1	0	0	2	0	1	1	0	0	0
5	4	1	0	0	2	1	1	1	0	0	0
6	5	1	0	0	2	1	2	1	0	0	0
7	5	1	0	0	2	1	3	1	0	0	0

Entonces, el vector contador, al final del proceso, tendrá los siguientes valores:

contador

1	0	0	2	1	3	1	0	0	0
0	1	2	3	4	5	6	7	8	9

El contenido final del vector contador significa que en el vector dígitos hay la siguiente información:

1 casilla con el valor 0

Ninguna casilla con el valor 1
Ninguna casilla con el valor 2
2 casillas con el valor 3
1 casilla con el valor 4
3 casillas con el valor 5
1 casilla con el valor 6
Ninguna casilla con el valor 7
Ninguna casilla con el valor 8
Ninguna casilla con el valor 9

El código de este programa se presenta en el cuadro 7.10. Se utiliza una función para la captura de los datos en el vector dígitos, otra para contar el número de cada dígito existente en el vector dígitos, otra función para mostrar los resultados y el bloque principal donde se definen inicialmente los vectores y se hacen los llamados a las funciones mencionadas.

Cuadro 7.10 Conteo de dígitos almacenados en un vector

```
 1.  ; Función para capturar
 2.  (define captura-vector
 3.    (λ (v)
 4.      (do ((i 0(+ i 1)))((>= i (vector-length v)))
 5.        (display "Digite valor para la posición ")
 6.        (display i)
 7.        (display " : ")
 8.        (vector-set! v i (read))
 9.       ); Fin do
10.   ); Fin lambda
11. ); Fin define
12.
13. ; Función para contar el número de dígitos en el
    vector v1
14. (define cuenta-vector
15.   (λ (v1 v2)
16.     (do ((i 0(+ i 1)))((>= i (vector-length
    v1)))
```

Cuadro 7.10 (Continuación)

```
17.         (vector-set! v2 (vector-ref v1 i) (+ 1
            (vector-ref v2 (vector-ref v1 i))))
18.       ); Fin do
19.     ); Fin de lambda
20. ); Fin de define
21.
22. ; Función para mostrar el contenido de los
    vectores
23. (define muestra-resultados
24.     (λ (v1 v2)
25.       (display "\nVector digitos: ")
26.       (display v1); Muestra todo el vector
27.       (newline)
28.       (do ((i 0(+ i 1)))((>= i (vector-length
                                    v1)))
29.         (display "Cantidad de ")
30.         (display i)
31.         (display " en dígitos : ")
32.         ;Muestra el contenido de una posición del
             vector
33.         (display (vector-ref v2 i))
34.         (newline)
35.       ); Fin do
36.     ); Fin lambda
37. ); Fin define
38.
39. ; Bloque principal
40. (begin
41.     (define digitos(make-vector 8))
42.     (define contador(make-vector 10))
43.     (captura-vector dígitos); Llama a la función
                                  captura-vector
44.     ; Llama a la función cuenta-vector
        (cuenta-vector digitos contador)
45.     ; Llama a la función muestra-resultad
        (muestra-resultados digitos contador)
46. )
```

Cuadro 7.10 (Continuación)

```
Digite valor para la posición  0 : 3
Digite valor para la posición  1 : 5
Digite valor para la posición  2 : 3
Digite valor para la posición  3 : 0
Digite valor para la posición  4 : 6
Digite valor para la posición  5 : 4
Digite valor para la posición  6 : 5
Digite valor para la posición  7 : 5

Vector digitos: #(3 5 3 0 6 4 5 5)
Cantidad de 0 en dígitos : 1
Cantidad de 1 en dígitos : 0
Cantidad de 2 en dígitos : 0
Cantidad de 3 en dígitos : 2
Cantidad de 4 en dígitos : 1
Cantidad de 5 en dígitos : 3
Cantidad de 6 en dígitos : 1
Cantidad de 7 en dígitos : 0
Cantidad de 8 en dígitos : 0
Cantidad de 9 en dígitos : 0
```

En el anterior programa la función que realiza el conteo de los dígitos almacenados en el vector *dígitos* es cuenta-vector en la cual utiliza el ciclo:

```
(do ((i 0(+ i 1))) ((>= i (vector-length v1)))
     (vector-set! v2 (vector-ref v1 i) (+ 1 (vector-
           ref v2 (vector-ref v1 i))))
)
```

Este ciclo lee posición a posición el vector v1 (dígitos) hasta que i sea mayor o igual que la longitud del vector v1; en cada iteración le asigna al vector v2 (contador) su valor anterior más uno. La posición del vector v2 que se actualiza corresponde al valor contenido en la posición leída del vector v1 así:

```
(vector-set! v2 (vector-ref v1 i) (+ 1 (vector-ref v2 (vector-ref v1 i))))
```

Vector contador

Posición a actualizarse

Suma 1 al vector v2

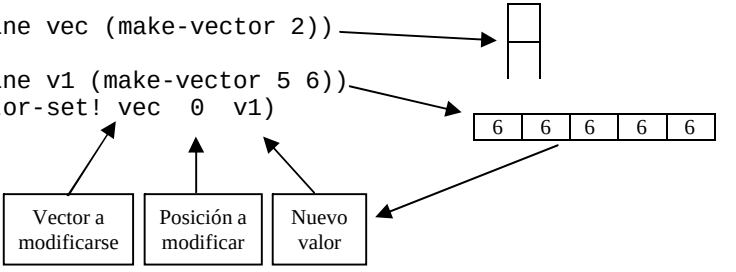
En la posición que almacena el vector v1 en la posición i

El contenido del vector `v1` en la posición `i` se convierte en la posición del vector `v2` que se va a actualizar, de esta manera cada posición del vector contador se convierte a su vez en un contador: la posición `0` sirve de contador de ceros, la posición `1` sirve de contador de unos, etc.

Ejemplo 7.10. Programa que crea un vector y en cada posición almacena otro vector

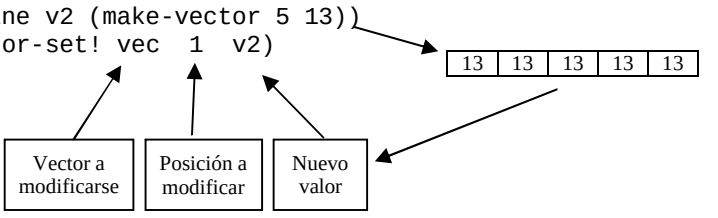
```
(define vec (make-vector 2))

(define v1 (make-vector 5 6))
(vector-set! vec  0  v1)
```



Para almacenar en la posición `1` de vec el vector `#(13 13 13 13 13)` se procede así:

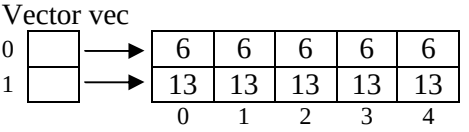
```
(define v2 (make-vector 5 13))
(vector-set! vec  1  v2)
```



Los elementos de un vector pueden ser de cualquier tipo, ya sean simples o compuestos. Por esto, para almacenar un vector dentro

de otro se procede de igual manera que para almacenar un dato simple, una vez se haya creado un vector éste puede utilizarse como una variable cualquiera y guardarse en una posición de otro.

El resultado final en el vector vec será:



El cuadro 7.11 presenta el programa que implementa un vector de vectores.

Cuadro 7.11 Vector de vectores

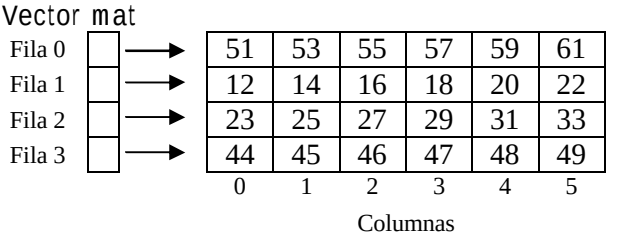
```
1.  ;Crea un vector que contiene un vector en cada
    posición
2.  (begin
3.    (define vec (make-vector 2))
4.    (define v1 (make-vector 5 6))
5.    (define v2 (make-vector 5 13))
6.    (vector-set! vec 0 v1)
7.    (vector-set! vec 1 v2)
8.    (display (vector-ref vec 0))
9.    (newline)
10.   (display (vector-ref vec 1))
11. )
```

```
#(6 6 6 6 6)
#(13 13 13 13 13)
```

Ejemplo 7.11. Programa que implementa un arreglo de dos dimensiones.

Los arreglos de una dimensión se conocen como vectores, mientras que los de dos o más se conocen como matrices. En Scheme se puede crear matrices de tantas dimensiones como sea necesario siguiendo la estrategia mostrada en el ejemplo anterior, es decir, colocando una arreglo como elemento de otro.

En el programa del cuadro 7.12 se crea una matriz de 4 filas y 6 columnas:



Para referirse a una de las posiciones de esta matriz se utiliza la función vector-ref, por ejemplo, para hacer referencia al dato de la fila 2 con columna 4, (valor 31) se hace de la siguiente manera:

```
(vector-ref (vector-ref mat 2)  4)
```

vector almacenado en la posición 2 del vector mat

De esta manera (vector-ref (vector-ref mat 2) 4) hace referencia a la posición 4 del vector obtenido en el paréntesis interno.

La función (vector-ref (vector-ref mat 2) 4) muestra en pantalla el valor 31.

El ejemplo 7.12 utiliza dos ciclos, el primero controla las filas con la variable f que inicia en 0 y se incrementa en uno hasta que sea igual o mayor a la longitud del vector mat, es decir (2). Para cada valor de f se ejecuta un ciclo interno controlado con la variable c que inicia en 0 y se incrementa en uno hasta que sea igual o mayor a la longitud del vector almacenado en la posición f del vector mat.

La longitud del vector mat es: (vector-length mat)

La longitud del vector contenido en una de las posiciones de mat es: (vector-length (vector-ref mat posicion))

Cuadro 7.12 Matriz de datos

```
1.  ;Crea y muestra un vector de vectores en forma de
    matriz
2.  (begin
3.    (define mat (make-vector 4))
4.    (define v0 #(51 53 55 57 59 61))
5.    (define v1 #(12 14 16 18 20 22))
6.    (define v2 #(23 25 27 29 31 33))
7.    (define v3 #(44 45 46 47 48 49))
8.    ;Almacena el vector v0 en la posición 0 de mat
9.    (vector-set! mat 0  v0)
10.   ;Almacena el vector v1 en la posición 1 de mat
11.   (vector-set! mat 1  v1)
12.   ;Almacena el vector v2 en la posición 2 de mat
13.   (vector-set! mat 2  v2)
14.   ;Almacena el vector v3 en la posición 3 de mat
15.   (vector-set! mat 3  v3)
16.   (display "Impresión de la matriz (")
17.   ;longitud de mat
18.   (display (vector-length mat))
19.   (display " filas por ")
20.   ;longitud del vector contenido en la posición 0 de
       mat
21.   (display (vector-length (vector-ref mat 0)))
22.   (display " Columnas)\n")
      (do ((f 0 (+ f 1))) ((>= f (vector-length
23.                            mat)));Ciclo para filas
                               (f)
24.       ;Ciclo para columnas (c)
        (do ((c 0 (+ c 1))) ((>= c (vector-length
25.                                          (vector-ref
                                             mat 0))))
26.       ;Muestra el contenido de la fila f con columna
           c
27.       (display (vector-ref (vector-ref mat f) c))
28.       (display " ")
29.     );fin do
30.     (newline)
31.   ); fin do
32. );fin begin
```

Cuadro 7.12 (continuación)

```
Impresión de la matriz (4 filas por 6 Columnas)
51 53 55 57 59 61
12 14 16 18 20 22
23 25 27 29 31 33
44 45 46 47 48 49
```

7.3.7 *vector-fill!*

La función vector-fill! almacena un mismo valor en cada una de las posiciones del vector. Su sintaxis es:

```
(vector-fill! vector obj)
```

Donde obj es el valor que se va a almacenar en todas las posiciones de vector.

A continuación se va a asignar a todas las posiciones del vector vec el valor 0 utilizando la función vector-fill!:

La expresión:

```
(define  vec  #(1 6 68 2 6 96))
```

crea el siguiente vector:

vec

1	6	68	2	6	96
0	1	2	3	4	5

La expresión:

```
(vector-fill!  vec  0)
```

deja el vector así:

vec

0	0	0	0	0	0
0	1	2	3	4	5

7.3.8 *vector?*

La función vector? sirve para determinar si un objeto es un vector. Retorna un valor booleano #t o #f dependiendo de si el objeto que se analice sea o no un vector.

La sintaxis de esta función es

```
(vector? obj)
```

Donde obj puede ser cualquier tipo de elemento, en el caso de ser un vector la función retorna #t, de lo contrario retorna #f.

Ejemplo 7.12. Programa que evalúa diferentes objetos y determina si son de tipo vector.

En el programa del cuadro 7.13 se definen variables de diferente tipo: vector, entera, carácter y cadena; a cada una de ellas se la envía a la función *valida*, la cual, está diseñada para determinar si el objeto recibido es un vector o no lo es.

Cuadro 7.13 Programa para determinar si un objeto es de tipo vector

```
1.  (define valida
2.    (λ (obj)
          (if (vector? obj) ; Pregunta si el objeto
3.                          recibido, (obj), es un
                            vector
4.          (begin
5.            (display "\nEl objeto ")
6.            (display obj)
7.            (display " es un vector ")
8.          ) ; fin begin
9.          (begin
```

Cuadro 7.13 (Continuación)

```
10.          (display "\nEl objeto ")
11.          (display obj)
12.          (display " NO es un vector")
13.        ); fin begin
14.     ) ; fin if
15.    ) ; fin lambda
16.  ) ; fin define
17.  ; Bloque principal
18.  (begin
19.    (define vec #("Pasto" "Popayán" "Cali" "Bogotá"))
20.    (define x 10)
21.    (define y #\a)
22.    (define y 13.6)
23.    (define z "Julián")
24.    (valida vec) ; Llama a valida y le envía un vector
25.    (valida x) ; Llama a valida y le envía un entero
26.    (valida y) ; Llama a valida y le envía un carácter
27.    (valida z) ; Llama a valida y le envía una cadena
28.  ) ; fin begin
```

```
El objeto #(Pasto Popayán Cali Bogotá) es un vector
El objeto 10 NO es un vector
El objeto 13.6 NO es un vector
El objeto Julián NO es un vector
```

7.4 BÚSQUEDA DE UN ELEMENTO EN UN VECTOR

Para buscar datos en un vector se cuenta con dos métodos: búsqueda lineal y búsqueda binaria. La primera se puede aplicar en cualquier vector mientras que la segunda solo en vectores ordenados.

La búsqueda lineal consiste en recorrer el vector posición a posición y comprobar si en ella se encuentra el dato que se está buscando.

Ejemplo 7.13. Programa que busca un número en un vector e informa la posición en que se encuentra. Si no lo encuentra muestra el mensaje correspondiente.

El código se presenta en el cuadro 7.14. El programa implementa una función recursiva llamada búsqueda-lineal. Esta recibe como argumento el vector y la posición a comparar, si encuentra el dato devuelve la posición, sino verifica si faltan posiciones por verificar (posición mayor a 0) e invoca a nuevamente a la función. Si se revisan todas las posiciones y no se encuentra el dato devuelve -1.

Cuadro 7.14 Búsqueda lineal

```
1.  ;Esta función busca el dato b en un vector
2.  ;Si se encuentra el dato b retorna 0, si no lo
    encuentra retorna 1
3.  (define busqueda-lineal
4.    (λ (vec pos num)
5.      (if (= (vector-ref vec pos) num)
6.          pos; Retorna uno
7.          (if (= pos 0)
8.              -1
9.              (busqueda-lineal vec (- pos 1) num)
                 ;Llamado recursivo
10.         ) ; Fin if
11.      ) ; Fin if
12.   ); Fin lambda
13. ) ; Fin define
14.
15. ; Bloque principal
16. (begin
17.   (define dato 0)
18.   (define vec (vector 16 13 6 9 23 33))
19.   (display "Digite valor a buscar : ")
20.   (set! dato (read)); Captura dato a buscar
21.   (display "El valor ")
22.   (display dato)
23.   (define pos-dato (busqueda-lineal vec (- (vector-
                                    length vec) 1) dato))
24.   (if (>= pos-dato 0)
25.       (begin
26.        (display " está en la posición ")
27.        (display pos-dato))
28.       (display " NO se encuentra en el vector")
29.   ); Fin if
30. ) ; Fin begin
```

```
Digite valor a buscar : 9
El valor 9 está en la posición 3
Digite valor a buscar : 156
El valor 156 NO se encuentra en el vector
```

7.5. IMPLEMENTACIÓN DE UN PROGRAMA DE INVENTARIO

Para finalizar este capítulo se propone un ejemplo en el que se aplica todo tratado. Se trata de un programa para manejar el inventario y calcular el valor por pagar en la compra de prendas en un almacén de ropa, para ello, se utilizan 4 vectores correspondientes a: los nombres de los productos, la cantidad de cada uno de ellos, el precio y el descuento por producto.

En el cuadro 7.15 se muestra, por cada prenda, la cantidad existente en bodega, el precio por unidad y el descuento que se da a cada producto.

Cuadro 7.15. Descuentos por cantidad

#	Prenda	Cantidad en Bodega	Precio unidad	Descuento
1	Pantalón	250	45.000	0.10
2	Sudadera	220	75.000	0.05
3	Zapatos	150	185.000	0.15
4	Camibusos	130	45.000	0.08
5	Camisetas	400	25.000	0.13

Con esta información se diseñan los siguientes vectores:

Prendas

Pantalón	Sudadera	Zapatos	Camibusos	Camisetas
0	1	2	3	4

Cantidad

250	220	150	130	400
0	1	2	3	4

Precios

45000	75000	185000	45000	25000
0	1	2	3	4

Descuentos

0.10	0.05	0.15	0.08	0.13
0	1	2	3	4

La posición *n* de cada uno de los vectores está relacionada con una prenda, por ejemplo, la posición 0 de prendas, cantidad, precios y descuentos se refiere a la prenda pantalón; en prendas se almacena el nombre, mientras que en los otros vectores se almacena correspondientemente la cantidad en inventario, el precio por unidad y el descuento en la venta.

El código fuente de este programa se presenta en el cuadro 7.16 y utiliza las siguientes funciones:

Inicializa-inventario: Inicializa la cantidad de elementos de cada prenda existente en inventario con el valor digitado por el usuario.

Muestra-valores: Encargada de mostrar la información almacenada en los cuatro vectores mencionados

Actualiza-cantidad: Permite al usuario modificar la cantidad de una prenda existente en inventario.

Actualiza-precio: Permite al usuario modificar el precio de una prenda.

Factura-venta: Permite hacer una facturación, solicita inicialmente el nombre de la prenda, si existe, captura la cantidad, si esta es menor que la cantidad en almacén calcula el descuento y el valor a pagar por la prenda. El usuario puede seguir comprando artículos hasta que él lo considere, cuando termina se calcula el valor total a pagar. Hay que tener en cuenta que por cada venta la cantidad en inventario se disminuye.

Bloque principal: define vectores de trabajo y presenta un menú con las opciones de inicializar inventario, mostrar datos, actualizar cantidad, actualizar precio, realizar factura y terminar.

Cuadro 7.16. Programa para manejo de un inventario

```
1.  ; --- Muestra los elementos de los vectores prendas,
    cantidad, precios y descuentos ---
2.  (define muestra-valores
3.    (λ (prendas cantidad precios descuentos)
4.      (define i 0)
5.      (display   "\n**********      MUESTRA   DATOS   DE
                   PRODUCTOS  ********** \n")
6.      (do  ((i  0  (+  i  1)))  ((>  i  (-  (vector-length
                                          prendas) 1)))
7.        ;Muestra el contenido de la posición i del
           vector prendas
8.        (display (vector-ref prendas i))
9.        (display ": ")
10.       ;Muestra el contenido de la posición i del
           vector cantidad
11.       (display (vector-ref cantidad i))
12.       (display ", ")
13.       ;Muestra el contenido de la posición i del
           vector precios
14.       (display (vector-ref precios i))
15.       (display ", ")
16.       ;Muestra el contenido de la posición i del
           vector descuentos
17.       (display (vector-ref descuentos i))
18.       (newline)
19.     )))
20.
21. ; --- Proceso de facturación ---
22. (define factura-venta
23.   (λ (prendas cantidad precios descuentos)
24.     (define i 0)
25.     (define tot 0)
26.     (define cant 0)
27.     (define des 0)
28.     (define gt-des 0)
29.     (define gt-tot 0)
30.     (define opc null)
31.     (define pdto null)
32.     (define sw 1)
33.     (display  "\n**********   PROCESO  DE  FACTURACION
                  ********** \n")
34.     (= i 0)
35.     (let ciclo ((continuar #t))
36.       (when (boolean=? Continuar #t)
37.         (display "\n\nProducro a facturar: ")
```

```
38.          (set!  pdto  (read));Captura  valor  para  la
                                  variable pdto
39.          (set! sw (+ 1 0));Auxiliar en 1 indica que no
                               se encuntra el artículo
40.          (do ((i 0 (+ i 1))) ((> i (- (vector-length
                                    prendas) 1)))
41.            (when   (equal?   (vector-ref   prendas   i)
                                       pdto);if-1
42.                ;(begin
43.                  (set! sw (+ 0 0)); sw en 0 significa
                                que si existe el producto
44.                  (display "\nNo de prendas: ")
45.                  (set! cant (read))
46.                  (if  (>  cant  (vector-ref  cantidad
                                       i));if-2
47.                  (begin
48.                    (newline)
49.                    (display  "\nCantidad  superior  al
                                       stock: ")
50.                    (display (vector-ref cantidad i))
51.                  );fin begin
52.                  (begin
53.                    (vector-set! cantidad i (- (vector-
                        ref cantidad i) cant))
54.                    (display "\nSubtotal: ")
55.                    (display   (*   cant   (vector-ref
                          precios i)))
56.                    (display "\nDescuento: ")
57.                    (set!   des   (*   cant   (vector-ref
                                precios    i)    (vector-ref
                                descuentos i)))
58.                    (set! gt-des des)
59.                    (display gt-des)
60.                    (display "\nValor a pagar: ")
61.                    (set!  tot  (-  (*  cant  (vector-ref
                          precios i)) des))
62.                    (set! gt-tot tot)
63.                    (display gt-tot)
64.                  );fin begin
65.                  );fin if-2
66.              );fin when
67.           );fin do
68.          (when (equal? sw 1)
69.              (display "\nProducto desconocido")
70.          );fin when
71.          (display "\n\nOtro producto ? ")
```

```
72.          (set! opc (read))
73.          (if (equal? Opc 's)
74.            (ciclo #t)
75.            (ciclo #f)
76.          );fin if
77.        );fin when
78.      );fin let
79.    ); fin lambda
80.  );fin define
81.
82.  ; --- Inicializa la cantidad de prendas existentes en
     inventario ---
83.  (define inicializa-inventario
84.    (λ (prendas cantidad)
85.      (define val 0)
86.      (define i 0)
87.      (display "\n**********   INICIALIZA CANTIDAD EN
              INVENTARIO  ********** \n")
88.      ;Ejecuta ciclo hasta que i sea mayor que la
            longitud del vector prendas
89.      (do ((i 0 (+ i 1))) ((> i (- (vector-length
                                        prendas) 1)))
90.        (display "Valor inicial para : ")
91.        ;Muestra el contenido de la posición i del
            vector prendas
92.        (display (vector-ref prendas i))
93.        ;Asigna el valor digitado por el usuario a la
            casilla i del vector cantidad
94.        (vector-set! cantidad i (read))
95.      )))
96.
97.  ; --- Cambia la cantidad de la prenda que el usuario
     digite ---
98.  (define actualiza-cantidad
99.    (λ (prendas cantidad)
100.     (define pdto null)
101.     (define val 0)
102.     (define i 0)
103.     (display "\n**********  ACTUALIZA CANTIDAD DE UN
           PRODUCTO  ********** \n")
104.     (display "Producto a actualizar : ")
105.     (set! pdto (read));Captura valor para la variable
           proa
106.     (do ((i 0 (+ i 1))) ((> i (- (vector-length
              prendas) 1)))
107.       (when (equal? (vector-ref prendas i) pdto)
```

```
108.          (display "Cantidad actual de ")
109.          (display (vector-ref prendas i));Muestra la
                                        posición i de prendas
110.          (display " : ")
111.          (display (vector-ref cantidad i));Muestra la
                      posición i de cantidad
112.          (newline)
113.          (display "Nueva cantidad : ")
114.          (vector-set! cantidad i (read));Captura valor
                  y se lo asigna a la posición i del
                  vector cantidad
115.        ))))
116.
117. ; --- Cambia el precio de la prenda que el usuario
     digite ---
118. (define actualiza-precio
119.   (λ (prendas precios)
120.     (define pdto null)
121.     (define val 0)
122.     (define i 0)
123.     (display "\n**********   ACTUALIZA PRECIO DE UN
             PRODUCTO  ********** \n")
124.     (display "Producto a actualizar : ")
125.     (set! pdto (read));Captura valor para la variable
            proa
126.     (do  ((i  0  (+  i  1)))  ((>  i  (-  (vector-length
             prendas) 1)))
127.       (when (equal? (vector-ref prendas i) pdto)
128.         (display "Precio actual de ")
129.         (display (vector-ref prendas i))
130.         (display " : ")
131.         (display (vector-ref precios i))
132.         (newline)
133.         (display "Nuevo precio : ")
134.         (vector-set! precios i (read));Captura valor
             y se lo asigna a la posición i del vector
             cantidad
135.       ))))
136.
137. ; --- Bloque principal ---
138. (begin
139.   (define prendas #("Pantalón" "Sudadera" "Zapatos"
         "Camibusos" "Camisetas"))
140.   (define cantidad (vector 250 220 150 130 400))
141.   (define precios (vector 45000 75000 185000 45000
         25000))
```

```
142.   (define  descuentos  (vector  0.10  0.05  0.15  0.08
        0.13))
143.   (define opcion null)
144.   (define cant 0)
145.   (let ciclo ((continuar #t))
146.     (when (boolean=? continuar #t)
147.         (display "\n\nMENU DE OPCIONES \n")
148.         (display "1. Inicializa inventario\n")
149.         (display "2. Listar productos\n")
150.         (display   "3.   Actualiza   cantidad   por
              producto\n")
151.         (display    "4.    Actualiza    precio    por
                 producto\n")
152.         (display "5. Factura venta\n")
153.         (display "6. Salir\n")
154.         (display "\n Cuál es su opción ?")
155.         (set! opcion (read))
156.         (case opcion
157.           ((1)     (inicializa-inventario     prendas
               cantidad) (ciclo #t))
158.           ((2)   (muestra-valores   prendas   cantidad
               precios descuentos) (ciclo #t))
159.           ((3) (actualiza-cantidad prendas cantidad)
               (ciclo #t))
160.           ((4)  (actualiza-precio  prendas  precios)
                (ciclo #t))
161.           ((5)   (factura-venta   prendas   cantidad
                   precios descuentos) (ciclo #t))
162.           ((6) (ciclo #f))
163.         )
164.     )))
```

CAPÍTULO 8

LISTAS

"Daría todo lo que sé,
por la mitad de lo que ignoro".
R. Descartes

Gran parte de la información que se maneja diariamente está constituida por conjuntos de datos que se modelan como listas. Algunos ejemplos son: la lista de útiles para cada año escolar, la lista de equipos de fútbol de Colombia, la lista de frutas, la lista de municipios, la lista de estudiantes, la lista de productos en un inventario, la lista de empleados de una empresa, etc.

Una lista es un conjunto de elementos que pueden ser del mismo o de diferentes tipos de datos, agrupados bajo una misma denominación (nombre de la lista) o sin un nombre determinado; su manejo se hace por medio de funciones ya establecidas por el lenguaje Scheme que permiten entre otras las tareas de: crear, adicionar, eliminar, buscar, invertir o filtrar datos.

En Scheme las listas son una poderosa herramienta que permiten manipular diferente tipo de información, puede trabajar con listas numéricas, de caracteres, de cadenas, de símbolos, incluso puede tener listas conformadas por otras listas y también listas que mezclen todos los tipos de datos anteriores.

Scheme cuenta con un amplio conjunto de funciones internas para realizar operaciones como: crear listas, adicionar, eliminar, buscar, ordenar y filtrar sus elementos. En este capítulo se explica el

manejo de listas, las principales funciones de Scheme y abundantes ejemplos que ayudarán a comprender el tema.

8.1 MANEJO DE LISTAS CON SCHEME

Para iniciar el manejo de listas lo primero que debe hacerse es definir la lista, en ella se le asigna un nombre y se la crea, ya sea vacía o compuesta con los elementos que el programador le asigne.

La expresión de creación de una lista tiene la forma:

```
(define nombre-lista  '(obj1 obj2 obj3….objn))
```

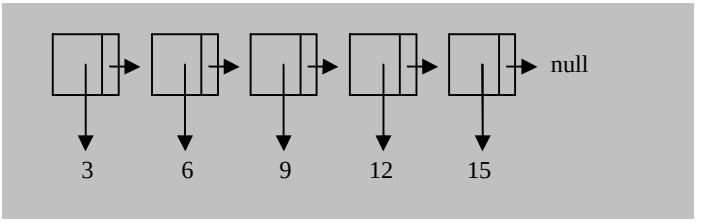
Donde *nombre-lista* es el nombre que el programador le da a la lista según su gusto y conveniencia; los elementos *obj1*, *obj2*, *obj3*,...*objn* son los componentes de la lista. Cada objeto puede ser un dato de diferente tipo, por ejemplo, una lista puede estar integrada por un valor numérico, un carácter, una cadena, un símbolo u otra lista. El apóstrofe (') o la función quote sirve para prevenir la evaluación de un símbolo

Ejemplo 8.1. Lista con los múltiplos de 3 menores o iguales que 15. Para crear esta lista se escribe la expresión:

```
(define multiplos3 '(3 6 9 12 15))
```

La forma de la estructura creada es similar a la figura 8.1.

Figura 8.1 Lista múltiplos de 3



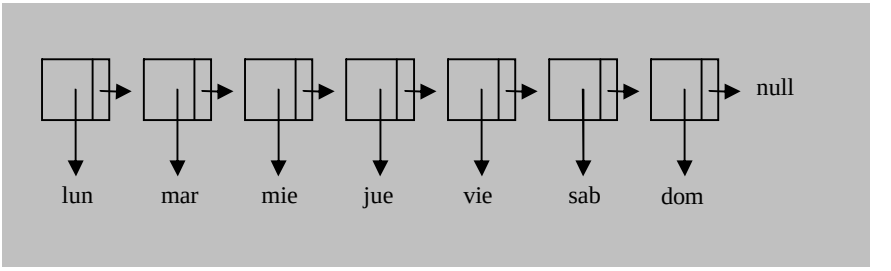
Ejemplo 8.2. Lista con los días de la semana

La expresión:

```
(define dias-semana '(lun mar mie jue vie sab dom))
```

crea una lista tal como se muestra en la figura 9.3.

Figura 8.2 Lista días-semana



El elemento *null* indica el final de la lista y será muy útil en las operaciones sobre listas por cuanto permitirá saber si aún existen elementos o se ha llegado al final de la lista o si la lista está vacía.

Se pueden manejar listas de números, de cadenas, de caracteres, de varios tipos de datos e incluso listas que contienen a su vez otras listas.

Las listas pueden estar asociadas a un identificador o pueden ser anónimas dependiendo del uso que se desee darles.

Una lista anónima tiene la forma:

```
‘(lunes martes miércoles jueves viernes sabado domingo)
```

La misma lista con identificador sería:

```
(define dias ‘(lunes martes miércoles jueves
viernes sabado domingo))
```

En el cuadro 8.1 se definen y muestran varias listas con elementos de diferente tipo.

Cuadro 8.1 Definición de listas

```
 1.   (begin
 2.    ;Lista múltiplos del 3
 3.    (define multiplos3 '(3 6 9 12 15))
 4.    (display multiplos3)
 5.    (newline)
 6.
 7.    ;Lista días de la semana
 8.    (define dias-semana '(lun mar mie jue vie sab dom))
 9.    (display dias-semana)
10.    (newline)
11.
12.    ;Lista solicitudes de impresión
13.    (define solicitudes-imp '((a b) c (d e) f))
14.    (display solicitudes-imp)
15.    (newline)
16.
17.    ;Lista de vocales
18.    (define vocales '(a e i o u))
19.    (display vocales)
20.    (newline)
21.
22.    ;Lista de números pares
23.    (define pares '(2 4 6 8 10))
24.    (display pares)
25.    (newline)
26.
27.    ;lista de cadenas
28.    (define nombres '(Linda María Luz))
29.    (display nombres)
30.    (newline)
31.
32.    ;Lista de números reales
33.    (define num-reales '(3.5 4.89 12.34))
34.    (display num-reales)
35.    (newline)
36.
37.    ;Lista de listas
38.    (define lis '((1 2 ) 3 (4 5 6)))
```

Cuadro 8.1 (Continuación)

```
39.    (display lis)
40.    (newline)
41.
42.    ;Lista con tipos de datos diferentes
43.    (define mixta '(13 Julián 6.7))
44.    (display mixta)
45.    (newline)
46.
47.    ;Lista vacía
48.    (define nomina (list null))
        ; list es una función que crea una lista con los
        argumentos que recibe, en este caso el argumento es null
49.    (display nomina)
50.  )
```

```
(3 6 9 12 15)
(lun mar mie jue vie sab dom)
((a b) c (d e) f)
(a e i o u)
(2 4 6 8 10)
(Linda María Luz)
(3.5 4.89 12.34)
((1 2) 3 (4 5 6))
(13 Julián 6.7)
(())
```

8.2 FUNCIONES PARA MANEJO DE LISTAS

El lenguaje de programación Scheme tiene diferentes funciones que permiten manipular una lista, mediante ellas se puede acceder a una lista, adicionar elementos, concatenar listas, calcular la longitud, buscar y filtrar datos, etc.

Las funciones para manejar las listas están predefinidas en el lenguaje Scheme y pueden ser utilizadas en cualquier parte del programa; entre las más importantes se encuentran las relacionadas en el cuadro 8.2.

Cuadro 8.2 Funciones para manejo de listas

Función	Descripción
car y first	Retornan la cabecera de la lista
cdr y rest	Retornan la cola de la lista
Length	Calcula la longitud de la lista
Cons	Añade un elemento al inicio de la lista
List	Crea una lista con los argumentos que recibe
Append	Concatena una segunda lista al final de la primera
member, memq y memv	Comprueban si un elemento es un miembro de una lista dada.
Reverse	Retorna una sublista resultado de invertir la lista inicial
Map	Aplica una función a todos los elementos de una lista
null y empty	Determina si una lista está vacía o no
Equal	Comprueba si un objeto es igual o equivalente a otro
list?	Determina si un objeto es una lista o no lo es
remove, remq y remv	Remueven un elemento de la lista
for-each	Aplica una función a cada elemento de una lista
Filter	Retorna una sublista compuesta por los elementos que coinciden con un predicado comparativo
Findf	Sirve para buscar un elemento de la lista
Memf	Obteniene una sublista que inicia en un valor que coincida con la condición de un predicado comparativo.
Sort	Ordena una lista ascendente o descendentemente
list-ref	Retorna el k-ésimo elemento de una lista
list-tail	Retorna una sublista que inicia en el k-ésimo elemento de la lista original.

8.2.1 Funciones *car* y *first*

Las funciones car y first son sinónimas; es decir, realizan la misma acción, retornan el primer elemento de una lista. Si la lista está vacía generan error de ejecución.

La sintaxis de estas funciones es:

```
(car lista)
```

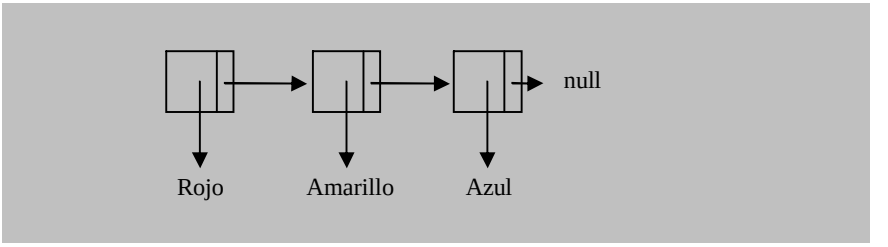
```
(first lista)
```

Por ejemplo, si se ha definido la lista de colores primarios:

```
(define colores-primarios '(Rojo Amarillo Azul)
```

Se tendría una lista como se muestra en la figura 8.3.

Figura 8.3 Lista colores-primarios



Al evaluar alguna de estas expresiones:

```
(car colores-primarios)
```

```
(first colores-primarios)
```

El resultado será: Rojo

Ejemplo 8.3. Programa que crea una lista de colores primarios y muestra el primer elemento de la lista. El código se muestra en el cuadro 8.3.

Cuadro 8.3. Programa que crea una lista de colores primarios

```
1.  (begin
2.    (display "Colores primarios \n")
3.    (define colores-primarios '(Rojo Amarillo Azul))
4.    (display (car colores-primarios))
5.  )
```

```
Colores primarios
Magenta
```

8.2.2 Funciones *cdr* y *rest*

Las funciones `cdr` y `rest` son sinónimas, retornan la cola de una lista, es decir, retornan una sublista compuesta por los elementos de la lista original, excepto el primero, que se obtiene con las funciones explicadas en 8.2.1..

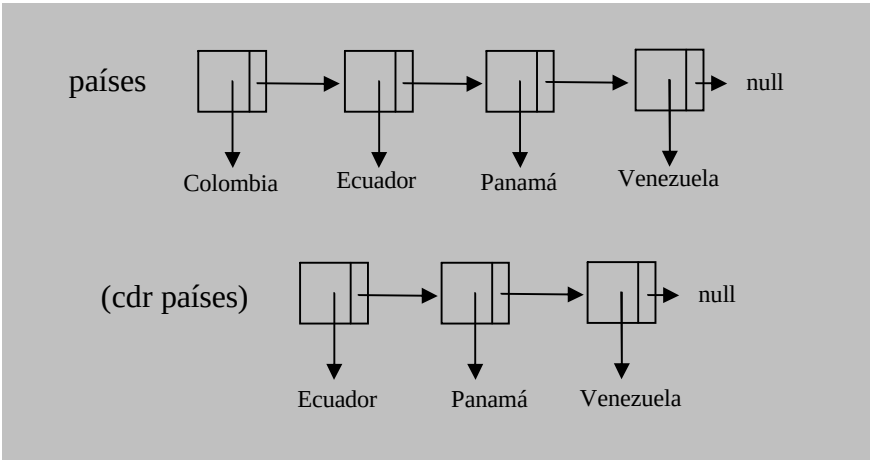
Ejemplo 8.4 Programa que crea una lista de países utilizando las funciones cdr y rest.

```
(define paises '(Colombia Ecuador Panamá
Venezuela))

(cdr paises)
```

Se tendrá como resultado de la función cdr o de la función rest una sublista integrada por los países de Ecuador, Panamá y Venezuela; en la figura 8.4 se puede observar el comportamiento de estas funciones.

Figura 8.4 Funciones *cdr* y *rest*



En el cuadro 8.4 se muestra la implementación de la lista países en Scheme.

Cuadro 8.4. Programa ejemplo de *cdr* y *rest*

```
1.  (begin
2.    (define paises '(Colombia Ecuador Panamá
    Venezuela))
3.    (display "\n Lista paises: ")
4.    (display paises)
5.    (display "\n (car paises): ")
6.    (display (car paises))
7.    (display "\n (cdr paises): ")
8.    (display (cdr paises))
9.    (display "\n (rest paises): ")
10.   (display (rest paises)))
```

```
 Lista paises: (Colombia Ecuador Panamá Venezuela)
 (car paises): Colombia
 (cdr paises): (Ecuador Panamá Venezuela)
 (rest paises): (Ecuador Panamá Venezuela)
```

Ejemplo 8.5. Programa que crea una lista y muestra su contenido aplicando recursividad.

A pesar de que la función display sobre una lista muestra todos sus elementos, en muchas ocasiones se necesita mostrarlos uno por uno; tal es el caso en que se requiera mostrar cada elemento en una línea diferente o cada uno acompañado de un texto. En el cuadro 8.5 se desarrolla una función que imprime elemento por elemento de forma recursiva, para ello, se debe entender que el único valor que se puede mostrar en la lista es el primero, obteniéndolo con la función car.

```
(display (car lista))
```

Esta expresión muestra el primer elemento de la lista, para mostrar el segundo se utiliza la función cdr para tomar la sublista restante y se invoca la función recursiva enviando esta nueva lista como parámetro. Estas operaciones se repetirán hasta que se obtenga una lista vacía.

Cuadro 8.5 Impresión recursiva de una lista

```
1.  (define muestra-lista
2.    (λ (lst)  ; Recibe como parámetro la lista lst
3.      (when(not(null? lst))  ; Cuando la lista NO esté
                               vacía haga
4.           (display (car lst))  ; Muestra la cabecera
5.           (newline)  ; Salta línea
6.           (muestra-lista (cdr lst)) ; Hace llamado
                     recursivo enviando la cola de lst
7.       ) ; Fin when
8.     )  ; Fin lambda
9.  ) ; Fin función
10.
11.  (begin
12.    (display "Países de la gran Colombia \n")
13.    (define paises '(Colombia Ecuador Panamá
     Venezuela))
14.    (muestra-lista paises)
15.  )
```

```
Países de la Gran Colombia
Colombia
Ecuador
Panamá
Venezuela
```

Ejemplo 8.6. Programa para mostrar una lista en orden inverso

Un algoritmo muy interesante donde se aplica la recursividad y las funciones car y cdr consiste en mostrar la lista de forma inversa, es decir, desde el último elemento hasta el primero.

Igual que en el ejemplo 8.5, el 8.6 recorre la lista haciendo llamados recursivos y enviando como parámetro la cola de la lista recibida. La diferencia se encuentra en que muestra en pantalla la cabecera de la lista después del llamado recursivo, esto hace que primero recorra la lista y al final cuando realice su proceso de retorno imprima, de tal manera que lo primero en mostrar será el último elemento, luego retorna e imprime el penúltimo y así sucesivamente hasta llegar al primer llamado de la función.

Cuadro 8.6 Impresión inversa de una lista

```
1.  (define muestra-lista-inv  ; Función muestra-lista
2.    (λ (lst)  ; Recibe como parámetro la lista lst
3.      (when(not(null? Lst
4.           (muestra-lista-inv (cdr lst))
5.           (display (car lst))
6.           (newline)
7.       ) ; Fin when
8.     )  ; Fin lambda
9.  ) ; Fin función
10.
11.  (begin
12.    (display "Países de la gran Colombia \n")
13.    (define paises '(Colombia Ecuador Panamá Venezuela))
14.    (muestra-lista-inv paises)
15.  )
```

```
Países de la Gran Colombia
Venezuela
Panamá
Ecuador
Colombia
```

Ejemplo 8.7. Programa para sumar los elementos de una lista numérica

En este ejemplo se suma el primer elemento más la sumatoria de la cola, para obtener el resultado de la sumatoria de la cola se suma nuevamente el primer elemento de ésta más la sumatoria de su cola y así sucesivamente hasta llegar al final de la lista, esto implica una función recursiva tal como se indica en el cuadro 8.7.

Cuadro 8.7 Sumatoria de los elementos de una lista

```
1.  (define suma-lista ; Función suma-lista
2.    (λ (lst) ; Recibe como parámetro la lista lst
3.      (if(null? lst) ; Si la lista está vacía haga:
4.           0 ; Por verdadero, Retorna 0
              (+ (car lst) (suma-lista (cdr lst))) ; Por
5.              falso, suma la cabecera de la lista más
                la sumatoria de la cola.
6.       ) ; Fin if
7.     )  ; Fin lambda
8.  ) ; Fin define
9.
10. (begin
      (display "Sumatoria de los elementos de una lista
11.                \n")
12.   (define numeros '(15 25 35 45))
13.   (display "La sumatoria de: ")
14.   (display numeros)
15.   (display " es igual a: ")
16.   (display (suma-lista numeros))
17. )
```

```
Sumatoria de los elementos de una lista
La sumatoria de: (15 25 35 45) es igual a: 120
```

Ejemplo 8.8. Programa para identificar el valor menor de una lista de números.

Para solucionar este problema es necesario comparar el valor de la cabecera con el valor menor del resto de la lista, pero este valor no lo puede determinar aún, para hacerlo envía la cola de la lista como argumento de la misma función recursiva, al recibirla nuevamente compara su elemento cabecera con el menor del resto de la lista recibida, igual que en la primera ocasión este valor no se puede determinar aún, de tal manera que nuevamente se repite el proceso hasta que en la lista recibida solo exista un elemento, al suceder

esto retorna este valor y se inicia el proceso de retorno en todos los llamados recursivos que quedaron pendientes, cada valor retornado es comparado con el elemento cabecera y el menor de estos se retorna al llamado anterior para ser comparado con la cabecera de la lista anterior hasta llegar al primer elemento de la lista original en cuyo caso el menor de los dos es el menor de toda la lista. El código de este programa se presenta en el cuadro 8.8.

Cuadro 8.8. Programa para encontrar el menor valor en una lista

```
1.  (define menor-lista ; Función menor-lista
2.   (λ (lst) ; Recibe como parámetro la lista lst
3.      (if (not (null? lst)) ; Si la lista NO está vacía haga:
4.          (if (null? (cdr lst)) ; Por verdadero,  Si la cola está vacía haga:
5.              (car lst) ; Por verdadero, retorna la cabeza de la lista
6.               ; Si el primer elemento es menor o igual a los elementos de la cola haga:
7.              (if (<= (car lst) (menor-lista(cdr lst)))
8.                (car lst) ; Por verdadero, retorna el primer elemento de la lista
9.                (menor-lista (cdr lst));  De lo contrario,  retorna valor menor de la cola
10.           ) ; Fin if
11.         ) ; Fin if
12.      ) ; Fin if
13.    ) ; Fin lambda
14. ) ; Fin función
15.
16. (begin
17.    (display "Valor menor de una lista \n")
18.    (define numeros '(12 23 6 13))
19.    (display "El valor menor en la lista: (")
20.    (display numeros)
21.    (display "), es igual a: ")
22.    (display (menor-lista numeros)))
Valor menor de una lista
El valor menor en la lista: ((12 23 6 13)), es igual a: 6
```

Es decir, la función llega al último elemento y retorna el valor para ser comparado con el penúltimo, entre los dos retorna el menor

para ser comparado con el antepenúltimo, el menor lo retorna para ser comparado con el anterior y así sucesivamente hasta comparar el menor con el primer elemento de la lista inicial.

8.3.3 Función *length*

La función length calcula la longitud de una lista, es decir, cuenta el número de elementos que la componen. La sintaxis de esta función es:

```
(length lista)
```

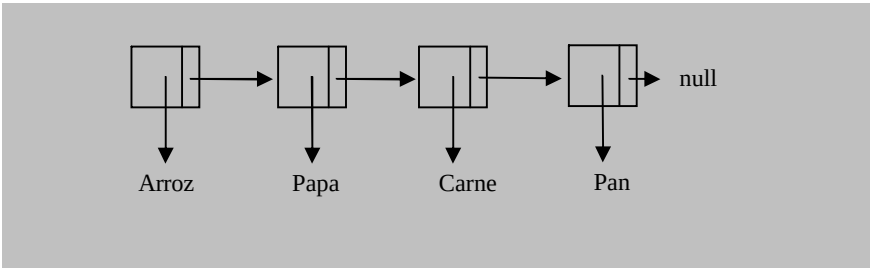
Ejemplo 8.9. Programa que crea una lista de elementos y muestra el número elementos que la componen

En este ejemplo se crea la lista utilizando la expresión:

```
(define alimentos '(arroz papa carne pan))
```

Define una lista como se muestra en la figura 8.5.

Figura 8.5 Lista de alimentos



Para obtener el número de productos de la lista alimentos se escribe:

```
(length alimentos)
```

Esta expresión devuelve como resultado: 4

El código completo se muestra en el cuadro 8.9

Cuadro 8.9 Conteo de los elementos de una lista

```
1.  (begin
2.    (display "Alimentos básicos \n")
3.    (define alimentos '(Arroz Papa Carne Pan))
4.    (display "La lista (")
5.    (display alimentos)
6.    (display "), contiene ")
7.    (display (length alimentos))
8.    (display " alimentos básicos.")
9.  )
Alimentos básicos
La lista ((Arroz Papa Carne Pan)), contiene 4 alimentos
básicos.
```

8.3.4 Función *cons*

La función cons inserta un elemento al inicio de la lista. Su sintaxis es:

```
(cons nuevo-elemento lista)
```

Ejemplo 8.10. Lista compuesta por los géneros de música: Pop, Rock y Salsa.

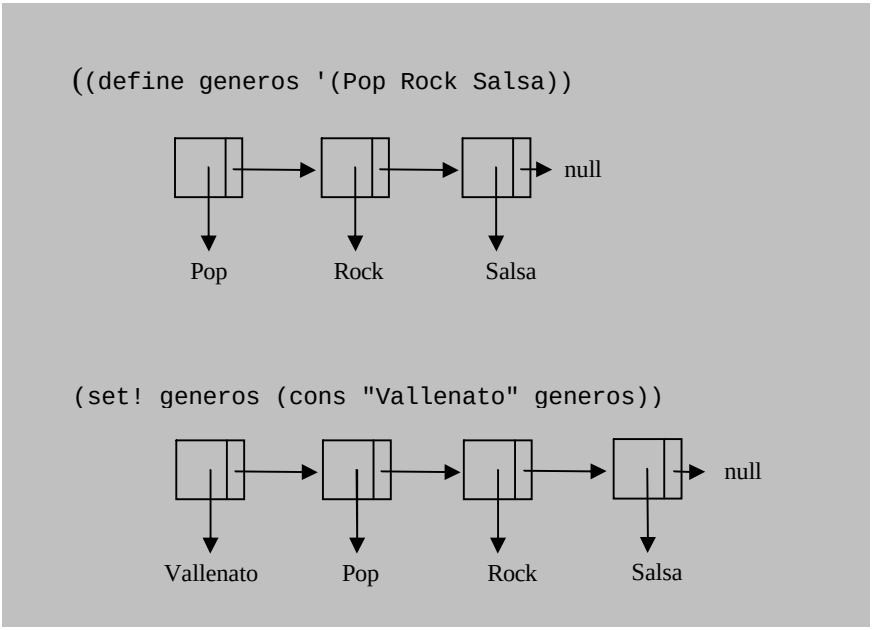
```
(define generos '(Pop Rock Salsa))
```

Utilizando la función cons se incluye el género Vallenato, con la particularidad de que lo coloca al inicio de la lista.

```
(set! generos (cons “Vallenato” generos))
```

El resultado es una lista con los elementos: Vallenato, Pop, Rock y Salsa, gráficamente se muestra en la figura 8.6 y el código en el cuadro 8.10.

Figura 8.6 Ejemplo de la función *cons*



Cuadro 8.10. Ejemplo de la función *cons*

```
    (begin
    (define generos '(Pop Rock Salsa))
    (display generos)
    (newline)
    (set! generos (cons "Vallenato" generos))
    (display generos))
```

```
(Pop Rock Salsa)
(Vallenato Pop Rock Salsa)
```

Ejemplo 8.11. Aplicación de la función cons en listas de diferentes tipos. El código se presenta en el cuadro 8.11

Cuadro 8.11 Aplicación de la función *cons*

```
 1.  (begin
 2.    ;Construcción de una lista de nombres
 3.    (define nombres '(Luis Consuelo))
 4.    (set! nombres (cons "Graciela" nombres))
 5.    (display nombres)
 6.    (newline)
 7.
 8.    ;Construcción de una lista de valores enteros
 9.    (define enteros '(10 20 35))
10.    (set! enteros (cons 5 enteros))
11.    (display enteros)
12.    (newline)
13.
14.    ;Construcción de la lista de empleados con código,
        nombre y tiempo de empleo
15.    (define empleados '()) ; Definición de la lista
        empleados vacía
16.    ;Adiciona un empleado
17.    (set! empleados (cons '(16 Julian 2) empleados))
18.    (display empleados) ; Muestra la lista empleados
19.    (newline)
20.
21.     ;Adiciona un empleado
22.    (set! empleados (cons '(12 Jenith 4) empleados))
23.    (display empleados) ; muestra la lista empleados
24.  )
```

```
(Graciela Luis Consuelo)
(5 10 20 35)
((16 Julian 2))
((12 Jenith 4) (16 Julian 2))
```

Ejemplo 8.12. Programa para crear y mostrar una lista de estudiantes.

El en cuadro 8.12 se muestra el código para leer seis nombres de estudiantes, colocarlos en una lista y luego mostrarlo. Observe que el primer nombre que se digita aparece al final de la lista y el último nombre capturado aparece al inicio de la lista.

Cuadro 8.12 Creación de la lista estudiantes con la función *cons*

```
 1.  (begin
 2.    (define estudiantes '()) ;Define la lista vacia
 3.    (define num-est 0) ;Define la variable num-est
 4.    ;Define la variable nom que servirá para capturar
        los nombres que se van a adicionar
 5.    (define nom null)
 6.    (display "LISTA DE ESTUDIANTES\n")
 7.     ;Inicia ciclo hasta que num-est sea menor que uno
 8.    (do ((num-est 6 (- num-est 1))) ((< num-est 1))
 9.       (display "Nombre Estudiante: ")
10.       (set! nom (read)) ;captura nombre
11.        ;adiciona elemento nom a la lista estudiantes
12.       (set! estudiantes(cons nom estudiantes))
13.     ) ; Fin do
14.    (display "La lista final es: ")
15.    (display estudiantes) ;Muestra la lista resultante
16.  ) ; Fin begin
```

```
LISTA DE ESTUDIANTES
Nombre estudiante: Katalina
Nombre estudiante: Julian
Nombre estudiante: Juan
Nombre estudiante: Yovana
Nombre estudiante: Natalia
Nombre estudiante: Paulo
La lista final es: (Paulo Natalia Yovana Juan Julian
Katalina)
```

Ejemplo 8.13 Programa para introducir datos en una lista numérica ordenada.

Para solucionar este problema se implementa una función recursiva que encuentre el lugar que le corresponde al número y lo inserte, de manera que la lista se construya ordenadamente. El código del programa y la función se presenta en el cuadro 8.13.

En cuadro 8.13 la función insertar-en-orden recibe una lista y un número para insertarlo en ella. En la primera invocación a la función la lista está vacía, en este caso simplemente inserta el primer elemento (línea 5). Si la lista tiene elementos la función verifica si el primer elemento es mayor que el número, lo que indica que éste debe insertarse en la primera posición (línea 7), sino invoca

nuevamente a la función y envía como parámetro la misma lista, pero sin el primer elemento (línea 8).

Cuadro 8.13. Adición de un número en una lista ordenada

```
1.  (define insertar-en-orden
2.    (λ (lista num)
3.      (if(null? Lista)
4.         (cons num lista)
5.         (if (<= num (car lista))
6.             (cons num lista)
7.             (cons (car lista) (insertar-en-orden (cdr
                                  lista) num))))))
8.  ;programa principal
9.  (begin
10.   (define mi-lista null)
11.   (let repetir ((rta 'S))
12.     (if(or(eqv? Rta 'S)(eqv? rta 's))
13.        (begin
14.          (display "\nNúmero: ")
15.          (set! mi-lista (insertar-en-orden mi-lista
                             (read)))
16.      (display "Lista ordenada: ")
17.          (display mi-lista)
18.          (display "\n Más números? S/N: ")
19.          (repetir (read))))))
```

```
Número: 10
Lista ordenada: (10)
 Más números? S/N: s

Número: 20
Lista ordenada: (10 20)
 Más números? S/N: s

Número: 40
Lista ordenada: (10 20 40)
 Más números? S/N: s

Número: 30
Lista ordenada: (10 20 30 40)
 Más numeros? S/N: s

Número: 5
Lista ordenada: (5 10 20 30 40)
 Más números? S/N: n
```

La función se ejecutará hasta que encuentre la posición que le corresponde al número, cuando lo haya insertado devuelve la lista que se genera. Cada vez que se recibe la lista proveniente de la ejecución recursiva se le coloca el elemento que se le había suprimido y se la devuelve al anterior espacio de activación.

En el programa principal se implementa un procedimiento recursivo let para insertar todos los números que el usuario desee, cada vez que inserta uno le muestra la nueva lista.

8.3.5 Función *list*

La función list crea una lista con los argumentos que recibe. Su sintaxis es:

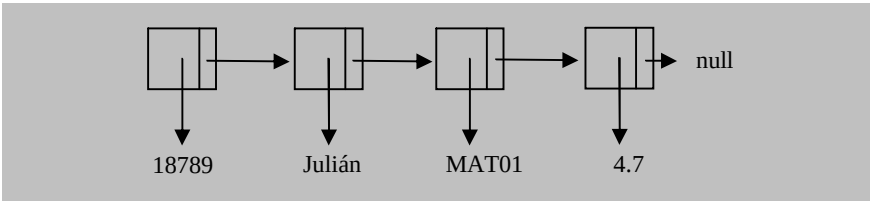
```
(list obj1 [obj2 ….. objn])
```

Con la función list se puede crear una lista de datos del mismo tipo o de diferente tipo, por ejemplo, se puede crear la lista para almacenar los datos de un estudiante: código, nombre, materia, nota.

```
(list 18789  'Julián  "MAT01"  4.7)
```

Esta expresión genera una lista como la que se muestra en la figura 8.7.

Figura 8.7. Ejemplo de la función *list*



Ejemplo 8.14. Programa que crea y muestra una listas con diferentes tipos de datos. Ver código en el cuadro 8.14.

Cuadro 8.14 Creación de listas con la función *list*

```
(begin
  ;Crea una lista con diferentes tipos de datos
  (display (list 'x 1 2 3.4 's '(x y z) "cc"))
  (newline)

  ;Crea una lista con diferentes tipos de datos, incluso el elemento '(x y z) es otra lista
  ;y el elemento (+ 3 4) es una operación, de la cual utiliza su resultado
  (display (list 'a (+ 3 4) 'c))
  (newline)

  ;Crea una lista de números enteros
  (display (list 10 20 30 40))
  (newline)

  ;Crea una lista conformada con dos lista y un entero
  (display (list '(10 20) '(30 40) 50))
  (newline)

  ;Crea una lista a partir de las listas definidas con anterioridad
  (define hombres '(Arturo Antonio Jesús Eduardo Pedro))
  (define mujeres '(Gladis Cristina))
  (display (list hombres mujeres))
  (newline)

  ;Crea una lista de números y se la asigna a lst
  (define lst '())
  (set! lst (list 1 2))
  (display lst)
)
```

```
(x 1 2 3.4 s (x y z) cc)
(a 7 c)
(10 20 30 40)
((10 20) (30 40) 50)
((Arturo Antonio Jesús Eduardo Pedro) (Gladis Cristina))
(1 2)
```

Como aprecia, una de las ventajas de ésta función es que se pueden crear listas compuestas por elementos que son el resultado del retorno de una función o de una operación como es el caso de (+ 3 4).

8.3.6 Función *findf*

La función findf sirve para buscar un elemento en una lista. Esta función compara su predicado con cada elemento de la lista y devuelve el primer elemento que encuentre. Si ningún elemento coincide, la función retorna null.

La sintaxis de la función es:

```
(findf  f lista)
```

Donde f puede ser cualquiera de los diferentes predicados que existen en el lenguaje Écheme.

Ejemplo 8.15. Aplicación de la función findf para identificar tipos de elementos. El código se presenta en el cuadro 8.15

Cuadro 8.15. Ejemplo de función *findf*

```
   (begin
1.   (define lst-int '(-32 2 -3 5 8 9 13 6 -15))
2.   (define lst '(#\f -3 1 5.6 -7.8 "b" a 'c 60 #\z
  '(Luis 12) 13))
3.   (define lst-num '(-32 2 -3 5.6 -7.8 80 13 6 -15))
4.   (define lst-car '(#\D #\@ #\b #\Z #\j #\A))
5.
6.   (display "a.De ")
7.   (display lst-int)
8.   (display " El primer número par es: ")
9.   (display (findf even? lst-int)); even solo recibe
   números enteros
10.  (newline)
11.
12.  (display "b.De ")
13.  (display lst)
14.  (display " El primer número es: ")
```

Cuadro 8.15 (Continuación)

```
15.   (display (findf number? lst))
16.   (newline)
17.
18.   (display "c.De ")
19.   (display lst-num)
20.   (display " El primer número positivo es: ")
21.   (display (findf positive? lst-num)); positive solo acepta números
22.   (newline)
23.
24.   (display "d.De ")
25.   (display lst)
26.   (display " El primer caracter es: ")
27.   (display (findf char? lst))
28.   (newline)
29.
30.   (display "e.De ")
31.   (display lst)
32.   (display " El primer símbolo es: ")
33.   (display (findf symbol? lst))
34.   (newline)
35.
36.   (display "f.De ")
37.   (display lst)
38.   (display " La primer lista es: ")
39.   (display (findf list? lst))
40.   (newline)
41.
42.   (display "g.De ")
43.   (display lst-car)
44.   (display " La primer minúscula es: ")
45.   (display (findf char-lower-case? lst-car)) ;char-lower-case solo acepta caracteres
46.   (newline)
47. )
```

```
a.De (-32 2 -3 5 8 9 13 6 -15) El primer número par es: -32
b.De (f -3 1 5.6 -7.8 b a 'c 60 z '(Luis 12) 13) El primer número es: -3
c.De (-32 2 -3 5.6 -7.8 80 13 6 -15) El primer número positivo es: 2
d.De (f -3 1 5.6 -7.8 b a 'c 60 z '(Luis 12) 13) El
```

```
primer caracter es: f
e.De (f -3 1 5.6 -7.8 b a 'c 60 z '(Luis 12)3) El primer
símbolo es: a
f.De (f -3 1 5.6 -7.8 b a 'c 60 z '(Luis 12) 13) La
primer lista es: 'c
g.De (D @ b Z j A) La primer minúscula es: b
```

Se ha explicado y presentado ejemplos de aplicación de las funciones más importantes para el manejo de listas. Muchas otras funciones que se presentaron en el cuadro 8.2 pueden usarse de manera similar para implementar las correspondientes tareas.

REFERENCIAS

APPLEBY D. y VANDEKOPPLE J. Lenguajes de programación: paradigma y práctica. 2 ed. México : McGraw-Hill, 1998. 492p.

BAASE, S. y GELDER, A. *Algoritmos computacionales: introducción al análisis y diseño*. 3 ed. México : Pearson Education, 2002.

BARZANALLANA, Rafael. Metodologías usadas en la ingeniería del software. Universidad de Murcia. España. 2005.

BRASSARD, G. y BRATLEY, T. Fundamentos de algoritmia. Madrid : Prentice Hall, 1997. 806 p.

CHAVES, Anivar. Algoritmos: pseudocódigo, diagramas de flujo y diagramas N-S. San Juan de Pasto : I. U. CESMAG, 2004. 297 p.

CLINGER, William, Semantics of Scheme, en BYTE, February, 1988. pp. 221-227.

COELLO, C. Scheme: lo pequeño es bello. Un Lenguaje de programación ideal para la enseñanza. Newsletters , Año 5, Vols. 15 y 16. 1996.

GUZMÁN, J. Introducción a la programación con Scheme. Editorial Tecnológica de Costa Rica, 2000.

JIMENEZ J. Programación Orientada a Objetos. Instituto politécnico Nacional. México. 2005.

JOYANES, L., Fundamentos de programación. Madrid : McGraw-Hill, 1988.

JOYANES, L. Progarmación en C++: algoritmos, estructuras de datos y objetos. Madrid : McGraw-Hill, 1999. 710 p.

KELSEY, R., CLINGER, W. y REES, J. Revised Report on the Algorithmic Languaje Scheme, s.l., 1998

LANGSAM, Y. y Otros. Estructuras de datos con C y C++. 2 ed. México : Prentice Hall, 1997. 670 p.

LIPSCHUTZ, S. Estructuras de datos. Meéxico : McGraw-Hill, 1993. 390 p.

LUCENA, M. Programación Declarativa. Jaén, Andalucía, 2003.

MCARTHY, J. LISP: Programming Manual. Cambridge. 1965

SCHILDT, Herbert. Aplique turbo C++. Madrid : MacGraw-Hill, 1993. 577 p.

SERÓN, F. MAGALLÓN, J. BALDASSARRI, S. Lenguajes de programación: Paradigmas. España : Universidad de Zaragoza, 2006.

SITARAM, D., Teach yourself Scheme in fixnum days. Boston : Northeaster University, 2004.

STEELE, G. L. LAMBDA: The Ultimate Imperative. AI Memo 353, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1976.

STEELE, G. L. RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization. Technical). Report 474, MIT Artificial Intelligence Laboratory. Massachusetts, 1978.

STEELE, G. L. SUSSMAN. J. G. Storage Management in a LISP-Based Processor. Caltech Conference on Very Large Scale Integration, Pasadena. 1979.

TUCKER, A. y NOONANN R. Lenguajes de programación, principios y paradigmas. McGraw Hill. 2003.

APENDICE A

LISTA DE EJEMPLOS

Ejemplo 3.1. Un programa que utiliza la función display para mostrar datos de diferente tipo. 62

Ejemplo 3.2. Un programa que utiliza la función *write* para mostrar datos de diferente tipo. 64

Ejemplo 3.3. Un programa que utiliza la función *read* para capturar el nombre del usuario y mostrar un saludo. 67

Ejemplo 3.4. Un programa que utiliza la función *read-char* para capturar un caracter. 68

Ejemplo 3.5. Un programa que utiliza la función *read-line* para capturar una cadena. 69

Ejemplo 3.6. Un programa que lee dos números, los suma y muestra el resultado. 70

Ejemplo 3.7. Un programa para calcular el área y el perímetro de un círculo. 71

Ejemplo 3.8. Un programa para calcular el valor futuro de una inversión de *x* capital a una tasa de interés *i* durante *n* periodos de tiempo. 71

Ejemplo 4.1. Función para obtener el valor absoluto de un número. 77

Ejemplo 4.2. Función para determinar si un número entero es par o impar. 78

Ejemplo 4.3. Función para convertir un dígito arábigo a su equivalente romano. 79

Ejemplo 4.4. En la tienda La Esquina se hace un descuento del 5% en compras de 10 o más unidades de un mismo producto.. 80

Ejemplo 4.5. En la empresa MIR se necesita un programa que permita calcular el valor de una bonificación para sus empleados. Esta es equivalente al 50% del sueldo mensual, pero para aquellos que hayan trabajado más de 20 años ésta se incrementa en un 30%.....81

Ejemplo 4.6. En la EPS Zdarobia se necesita un programa que calcule la cuota moderadora para sus usuarios. Esta tiene un valor de $ 10.000 para todos los usuarios, excepto para aquellos cuyos ingresos son mayores a 4 salarios mínimos, a quienes se les incrementa en un 1% sobre el valor su sueldo.....81

Ejemplo 4.7. Retomando el problema del ejemplo 4.4 se tiene: en la tienda La Esquina se hace un descuento del 5% en todas las compras, a menos que la cantidad comprada sea menor a 10 unidades.....82

Ejemplo 4.8. Función para determinar si un número *m* es múltiplo de *n*. 82

Ejemplo 4.9 Función para validar que el divisor sea diferente de cero antes de realizar una división.....84

Ejemplo 4.10. Función para determinar el auxilio de transporte de un empleado.....85

Ejemplo 4.11 Función para identificar el mayor de dos números. Considerando que los números sean diferentes entre sí, se evalúa si el primero es mayor que el segundo, si no se asume que el segundo es mayor que el primero.....86

Ejemplo 4.12 Función para realizar una división. Considerando que la división sobre 0 no está determinada se trata de evitar que el programa aborte al realizar el cálculo, para ello se valida el divisor, si este es 0 se devuelve un dato nulo. 86

Ejemplo 4.13. Un programa para calcular el área de un triángulo o de un círculo, según preferencia del usuario.....87

Ejemplo 4.14 Función que recibe el número del día y devuelve su nombre.....89

Ejemplo 4.15 Una función para determinar el número de días que tiene un mes.....90

Ejemplo 4.16. Un programa que mediante un menú efectúe las operaciones aritméticas básicas sobre dos números enteros.....91

Ejemplo 4.17. Una función para determinar si un año es bisiesto 93

Ejemplo 4.18. Una función que recibe tres números diferentes e identifica el mayor de ellos.....94

Ejemplo 4.19. La empresa Buen Patrón desea aumentar los sueldos de sus empleados de manera inversamente proporcional a la cantidad de salarios mínimos que devengan.....95

Ejemplo 5.1: Un programa que muestre los números del 1 al 10... 100

Ejemplo 5.2 Un programa que lea números desde el teclado y los sume. Cuando el usuario introduzca el 0 muestra el total y termina. 100

Ejemplo 5.3. Programa para mostrar los divisores de un número.. 101

Ejemplo 5.4: Un programa que lea *n* números y muestre el número mayor, el número menor y el promedio. Cuando se introduce el 0 no se leen más números. 102

Ejemplo 5.5. Un programa para calcular el promedio de cuatro notas validando que cada nota ingresada esté entre 0 y 5.0. 104

Ejemplo 5.6. Un programa para mostrar los números pares entre 2 y 20..... 106

Ejemplo 5.7. Un programa para mostrar la tabla de multiplica de un número cualquiera..... 106

Ejemplo 5.8. Un programa para calcular el promedio de notas de un grupo de *n* estudiantes. 107

Ejemplo 5.9. Una función para encontrar el Mínimo Común Múltiplo (MCM) de dos números. 107

Ejemplo 5.10. Un programa para determinar si un número es primo. 109

Ejemplo 7.1. Programa en Scheme para crear vectores de diferentes tipos y mostrar su contenido. 138

Ejemplo 7.2. Programa para crear vectores con define y make-vector y se mostrar el contenido de cada uno de ellos utilizando la función display. 142

Ejemplo 7.3. Crear vectores utilizando las funciones define y vector y mostrar su contenido. 143

Ejemplo 7.4. Programa para mostrar el contenido de un vector elemento por elemento. 148

Ejemplo 7.5. Programa para mostrar el contenido de un vector utilizando un ciclo do. 148

Ejemplo 7.6. Programa para mostrar el contenido de un vector aplicando recursividad. 149

Ejemplo 7.7. Mostrar el contenido de un vector en orden inverso. 150

Ejemplo 7.8. Programa que crea un vector y lo llena con datos capturados por el teclado. 153

Ejemplo 7.9. Un programa que lee dígitos y los guarda en un vector, luego cuenta cuántas veces aparece cada dígito. 153

Ejemplo 7.10. Programa que crea un vector y en cada posición almacena otro vector 158

Ejemplo 7.11. Programa que implementa un arreglo de dos dimensiones. 159

Ejemplo 7.12. Programa que evalúa diferentes objetos y determina si son de tipo vector. 163

Ejemplo 7.13. Programa que busca un número en un vector e informa la posición en que se encuentra. Si no lo encuentra muestra el mensaje correspondiente. 164

Ejemplo 8.1. Lista con los múltiplos de 3 menores o iguales que 15. 174

Ejemplo 8.2. Lista con los días de la semana 175

Ejemplo 8.3. Programa que crea una lista de colores primarios y muestra el primer elemento de la lista. 180

Ejemplo 8.4 Programa que crea una lista de países utilizando las funciones cdr y rest. 180

Ejemplo 8.5. Programa que crea una lista y muestra su contenido aplicando recursividad 181

Ejemplo 8.6. Programa para mostrar una lista en orden inverso.... 183

Ejemplo 8.7. Programa para sumar los elementos de una lista numérica 183

Ejemplo 8.8. Programa para identificar el valor menor de una lista de números. 184

Ejemplo 8.9. Programa que crea una lista de elementos y muestra el número elementos que la componen 186

Ejemplo 8.10. Lista compuesta por los géneros de música: Pop, Rock y Salsa. 187

Ejemplo 8.11. Aplicación de la función cons en listas de diferentes tipos. 189

Ejemplo 8.12. Programa para crear y mostrar una lista de estudiantes. 189

Ejemplo 8.13 Programa para introducir datos en una lista numérica ordenada. 190

Ejemplo 8.14. Programa que crea y muestra una listas con diferentes tipos de datos 193

Ejemplo 8.15. Aplicación de la función findf para identificar tipos de elementos. 194

Red de Universidades Regionales Latinoamericanas
UREL

Red de Universitaria de Investigación en Sistemas de Nariño
RUISNAR